

DTIC FILE COPY

RADC-TR-90-53, Vol II (of two)
Final Technical Report
May 1990



2

AD-A223 633

FORMAL VERIFICATION OF MATHEMATICAL SOFTWARE

Odyssey Research Assoc., Inc.

Sponsored by
Strategic Defense Initiative Office

DTIC
ELECTE
JUL 15 1990
S D D

~~DISTRIBUTION AUTHORIZED TO U.S. GOVERNMENT AGENCIES AND THEIR CONTRACTORS, CRITICAL TECHNOLOGY; May 90. OTHER REQUESTS FOR THIS DOCUMENT SHALL BE REFERRED TO RADC(COTC) GRIFFISS AFB, NY 13441-5700.~~

DISTRIBUTION STATEMENT A

Approved for public release
Distribution Unlimited

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Strategic Defense Initiative Office or the U.S. Government.

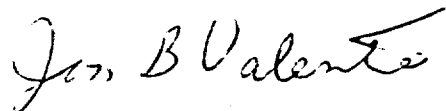
Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

90 07 28 016

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Services (NTIS) At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-90-53, Vol II (of two) has been reviewed and is approved for publication.

APPROVED:



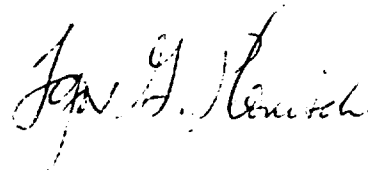
JON B. VALENTE
Project Engineer

APPROVED:



RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER:



IGOR G. PLONISCH
Directorate of Plans & Programs

Although this document references * documents listed on Pages 104 and 110, no limited information has been extracted.

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTC) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

FORMAL VERIFICATION OF MATHEMATICAL SOFTWARE

Stephen H. Brackin
Ian Sutherland

Contractor: Odyssey Research Assoc., Inc.
Contract Number: F30602-86-C-0116
Effective Date of Contract: 1 May 1986
Contract Expiration Date: 31 July 1989
Short Title of Work: Formal Verification of SDI
Mathematical Software
Period of Work Covered: May 86 - Jul 89

Principal Investigator: Stephen H. Brackin
Phone: (607) 277-2020

RADC Project Engineer: Jon B. Valente
Phone: (315) 330-3241

Approved for public release, distribution unlimited.

This research was supported by the Strategic Defense Initiative Office of the Department of Defense and was monitored by Jon B. Valente, RADC (COTC), Griffiss AFB NY 13441-5700 under Contract F30602-86-C-0116.

STATEMENT "A" per Jon Valente,
RADC/COTC, Griffiss AFB, NY 13441-5700
TELECON 7/3/90 VG

Accession For	
NTIS CRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By <i>per call</i>	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Project, Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE May 1990		3. REPORT TYPE AND DATES COVERED Final May 86 to Jul 89	
4. TITLE AND SUBTITLE FORMAL VERIFICATION OF MATHEMATICAL SOFTWARE				5. FUNDING NUMBERS C - F30602-86-C-0116 PE - 63223C PR - B413 TA - 03 WU - 03	
6. AUTHOR(S) Stephen H. Brackin, Ian Sutherland					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Odyssey Research Assoc., Inc. 301A Harris B Dates Drive Ithaca NY 14850-1313				8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Strategic Defense Initiative Office, Office of the Secretary of Defense Wash DC 20301-7100 Rome Air Development Center (COTC) Griffiss AFB NY 13441-5700				10. SPONSORING/MONITORING AGENCY REPORT NUMBER RADC-TR-90-53, Vol II (of two)	
11. SUPPLEMENTARY NOTES RADC Project Engineer: Jon B. Valente/COTC/(315) 330-3241					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) One of the major problems encountered in trying to formally verify the correctness of computer programs that use real arithmetic (hereinafter referred to as "mathematical programs") is that the mathematical properties of real arithmetic operations in computers are much more complicated and much harder to work with than the mathematical properties of the corresponding ideal mathematical operations. This occurs because the real number type implemented on a finite computer is not the same as the ideal, mathematical real number type. A finite machine can only represent finitely many different real numbers, whereas there are infinitely many ideal real numbers. The idea behind the theory of asymptotic computing is to develop techniques to prove that the accuracy of a mathematical program goes to infinity (e.g., larger and larger numbers of representation bits for mantissas and exponents used in binary floating point arithmetic). The theory of asymptotic computing, then, is essentially a general formalization of the notions of "accuracy" and "accuracy going to infinity". (Continued on Reverse)					
14. SUBJECT TERMS Formal verification, formal verification theory, verification, reals, verification, verification over reals, software verification, verification of mathematical programs.				15. NUMBER OF PAGES 226	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED				16. PRICE CODE	
18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED		19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED		20. LIMITATION OF ABSTRACT UL	

Block 13 Continued:

infinity", but without having to show how fast convergence happens (a major source of difficulty in numerical analysis).

(KR) ←

Contents

1	Introduction	1
2	Interval Analysis	5
2.1	Basic Definitions	5
2.2	Elementary Properties	7
2.3	Quantitative Results	8
2.3.1	FFT Multiplication Results	9
2.3.2	Interval Newton's Method Results	10
2.3.3	Summary	12
2.4	Interval Asymptotic Correctness	12
2.4.1	Asymptotic Correctness Definitions	13
2.4.2	Conjecture Problem Example	14
2.5	Matijasevich's Method	16
3	Floating-Point Arithmetic	21
3.1	Realism of the Asymptotic Model	21
3.2	Costs of Accuracy	23
3.3	IEEE and VAX Arithmetic	24
3.4	Floating-Point and Parallel Processing	25

4	Mathematics for Alternatives	28
4.1	Approximate Rational Arithmetic	28
4.2	Mediant Rounding	31
4.3	Standard Continued Fractions	32
4.4	Generalized Continued Fractions	35
4.5	Gosper's Algorithm	37
5	Alternatives in the Literature	45
5.1	Fixed-Slash and Floating-Slash	45
5.1.1	Representations of Numbers	46
5.1.2	Arithmetic Operations	49
5.1.3	Errors in Mediant Rounding	50
5.1.4	Empirical Results	52
5.1.5	Potential Applications	56
5.2	The LCF Representation System	58
5.2.1	The LCF Encoding	59
5.2.2	LCF Gap Sizes and Range	61
5.2.3	LCF Arithmetic	62
5.2.4	Possible Uses of Redundancy	65
5.3	Hybrid Floating-Point and Fixed-Slash	66
5.4	Variable-Length Exponents	68
5.5	Recurring-Digits Arithmetic	71
5.6	Hyper-Exponential Representations	73
5.7	Finite-Segment p -adic Representations	75
6	Constructive Reals	78
6.1	Definitions and Basic Properties	78
6.2	Advantages and Disadvantages	80

6.3	Implementations	82
6.4	Continued Fraction Results	83
6.4.1	Caliban Continued Fractions	83
6.4.2	Gosper's Algorithm	84
6.5	Boehm's Constructive Reals	88
7	Conclusions and Questions	90
7.1	Conclusions	90
7.2	Variable-Length-Exponents	93
7.3	Continued Fraction Questions	95
7.4	Integer Representations	96
8	Task Notes	100
8.1	Interval Probabilities	100
8.2	"Briggsian" Algorithms	101
8.3	Theoretical Floating-Point Speed Limits	101
8.4	Alternative Representations	101
8.5	Experiments on Boehm's Package	102
8.6	Aircraft Interception Example	103
8.7	Errata	103
A	IEEE Interval Arithmetic	111
A.1	Extended Real-Number Arithmetic	111
A.2	IEEE Floating-Point Arithmetic	112
A.3	Machine Interval Arithmetic	113
A.4	Semantics of Interval Operations	115
B	IEEE Interval Operations	117

C	FFT Multiplication Programs	134
C.1	IEEE Interval FFT Multiply	135
C.2	VAX Interval FFT Multiply	142
C.3	Scalar FFT Multiply	156
D	FFT Multiply Comparisons	163
E	Interval-Newton's Code	168
F	Interval-Newton's Results	173
G	A Correctness Difficulty	176
H	Caliban Continued Fractions	178
I	Gosper's Algorithm Code	183
I.1	Standard Continued Fractions	183
I.2	Generalized Continued Fractions	191
I.3	Sample Modifications	199

Chapter 1

Introduction

This is the final report for the Reals project's Task 5. It follows Chapter 3 of the Task's interim report [ORA88] and describes different methods for representing real numbers and performing operations on them in computers. It summarizes and evaluates significant ideas from the technical literature on computer arithmetic and interval analysis, relates results in interval analysis to the Reals project's work on asymptotic correctness, presents empirical results on naive and sophisticated interval algorithms and on VAX and IEEE-standard floating-point arithmetic, and lists questions for future research.

The representation ideas from the literature include one we consider superior to standard floating-point for command and control applications. The questions for future research include new proposals for representing the real numbers and an open theoretical question on representing the integers.

In our interim report, we classified error in computer calculations as either input error, modeling error, truncation error or computational error. We noted that different methods for representing real numbers and the elementary operations on them can only affect the computational error, which is often insignificant compared to other parts of the total error. We also noted that, in order to maintain reasonable computation speed and have stored values occupy reasonably small areas in computer memory, a representation system must discard information and hence introduce computational error.

We described interval analysis, a technique that maintains intervals whose endpoints are conservative upper and lower bounds on all input and com-

puted quantities, as a method for effectively eliminating error and replacing it with uncertainty, uncertainty reflected in the lengths of the intervals input or computed. Chapter 2 describes our subsequent work on interval analysis.

Chapter 2 gives empirical results on interval algorithms which show that naive interval analysis is unlikely to be useful but that more sophisticated interval algorithms can be surprisingly effective. The empirical results use a version of interval arithmetic that takes advantage of characteristics of IEEE-standard floating-point arithmetic [IEE85]. Chapter 2 briefly summarizes the Reals project's notion of asymptotic correctness [ORA87], describes a possible extension of this notion to interval arithmetic, and gives the results of our efforts to relate interval asymptotic correctness to scalar asymptotic correctness. Chapter 2 also describes a technique from Matijasevich [Mat85] for dealing with a basic defect of interval arithmetic that causes it to be overly conservative.

Our interim report noted that floating-point arithmetic has advantages that should not be neglected when considering alternative representations, and expressed the hope that using highly accurate floating-point arithmetic would be a practical way to make the asymptotic model accurate. Chapter 3 describes our subsequent work on floating-point arithmetic.

Chapter 3 gives examples showing the strengths and weaknesses of the asymptotic model, and discusses the costs and benefits of making it accurate. Chapter 3 summarizes results from the technical literature on the time, space and complexity costs of highly accurate floating-point arithmetic, particularly floating-point arithmetic compatible with the IEEE standard. Chapter 3 also describes results from the literature on on-line versions of floating-point arithmetic that facilitate doing parallel computation.

Our interim report reported that we would no longer pursue the Combinatorial Representation, which used algebraic topology, originally investigated by Task 5, but would instead study and evaluate other alternative representation systems for computer arithmetic given in the engineering literature. Chapters 4 and 5 describe results from the literature.

Chapter 4 describes mathematical ideas used by the alternative representation systems considered in Chapter 5. These ideas include approximate rational arithmetic, mediant rounding, standard and generalized continued fractions, and Gosper's algorithm for doing arithmetic on continued fractions.

Chapter 5 summarizes and comments on the following proposed representation systems: The fixed-slash and floating-slash representations by Matula and Kornerup [MK80,KM81,KM83a,MK85]; the binary-coded lexicographic continued fraction representation by Matula and Kornerup [MK83,KM85,KM87,KM88]; the hybrid fixed-slash and floating-point representation by Hwang and Chang [HC78]; the variable-length-exponent representation by Iri and Matsui [MI81]; the repeating-mantissa floating-point representation by Yoshida [Yos83]; the hyper-exponential representation by Olver and Clenshaw [Olv87]; and the finite p -adic representation by Gregory and Krishnamurty [GK84].

The approximate rational arithmetic systems from Matula and Kornerup show that our plan, given in our interim report, to only consider alternative representation systems as means for specifying the endpoints of intervals, was overly restrictive. Chapter 5's comments note for each representation whether it is suitable for representing the endpoints of intervals and whether it facilitates parallel computation. Chapter 5 also includes remarks about the empirical evidence from Matula and Ferguson [FM85] supporting the floating-slash representation system.

Our interim report described constructive-real representation systems that make it possible to do calculations to a user-specified or data-determined degree of accuracy. Our interim report noted that these systems could not be used for typical real-time computation applications because they do not discard enough information, but also noted that they might be useful for calculations requiring unpredictable amounts of accuracy in intermediate results. The constructive-real systems described in our interim report included a "lazily-evaluated continued fraction" system implemented by Jones [Jon84] and a "real as the limit of rationals" system by Boehm [Boe87]. Chapter 6 describes our work with constructive-real representation systems.

Chapter 6 lists general properties of constructive-real representation systems, including what we learned about their suitability for real-time applications. It gives Caliban programs for computing standard and generalized continued fraction expansions of rationals, and results on using Gosper's algorithm to perform arithmetic on standard and generalized continued fractions. It also gives additional information on Boehm's system of constructive-real arithmetic.

Chapter 7 summarizes the conclusions of this report. It gives our own suggestions for alternative representation systems and raises questions for future research, particularly a question about the theoretical limits of representations of the integers.

Finally, Chapter 8 ties up loose ends from Task 5. It notes plans or further studies that we were unable to carry out, and corrects two errors in our interim report.

Chapter 2

Interval Analysis

This chapter summarizes our work on interval analysis. It first defines interval analysis and terminology used later in the chapter, then reviews elementary relationships between interval and scalar arithmetic. It next gives quantitative results comparing naive interval computations to scalar ones, and contrasting the accuracy of naive and sophisticated interval algorithms. The chapter then describes the results of our efforts to define a notion of asymptotic correctness for interval algorithms and to relate proofs of interval and scalar asymptotic correctness. Finally, the chapter describes a proposal by Matijasevich [Mat85] for avoiding one of the major problems with interval analysis.

2.1 Basic Definitions

Interval analysis maintains exact bounds on the absolute error in all data values, and treats quantities as being known only to belong to intervals. The operations of *interval arithmetic* act on intervals and produce intervals that contain the results of the corresponding operations on *all* real numbers contained in the original intervals.

We will refer to real numbers as *scalar values*, to operations on real numbers as *scalar operations*, and to programs that use a fixed number representation system's values and operations — e.g., IEEE-standard double-precision

floating-point arithmetic -- instead of intervals and interval operations as *scalar programs*. Let \mathbf{M} be the set of values representable in the fixed representation system. If the endpoints (possibly $+\infty$ or $-\infty$) of an interval are in \mathbf{M} , call the interval *machine-representable*.

Let a *rounding* be a function $\square : \mathbf{R} \cup \{+\infty, -\infty\} \rightarrow \mathbf{M}$ satisfying:

$$\begin{aligned} \forall x \in \mathbf{M} \quad (\square x = x) \\ \forall x, y \in \mathbf{R} \quad (x \leq y \Rightarrow \square x \leq \square y) \end{aligned}$$

Call a rounding \square *upwardly directed* if $\square x \geq x$ and *downwardly directed* if $\square x \leq x$. Let \uparrow and \downarrow be upwardly and downwardly directed roundings, respectively. Define a *conservative rounding* on intervals by

$$\uparrow A = \uparrow [a_1, a_2] = [\downarrow a_1, \uparrow a_2].$$

For $\star \in \{+, -, \cdot, /\}$ and intervals A and B , if $a \star b$ is defined for every $a \in A$ and $b \in B$, define the interval operation $A \star B$ by letting

$$A \star B = \{a \star b \mid a \in A, b \in B\},$$

and let $A \star B$ be undefined otherwise. Define the *machine interval arithmetic operations* for $\star \in \{+, -, \cdot, /\}$ on machine-representable intervals A and B by letting

$$A \star_M B = \uparrow (A \star B)$$

whenever the resulting interval is defined, and letting it be undefined otherwise. We will refer to programs using machine-representable intervals and machine interval arithmetic operations, which we will usually refer to as intervals and interval arithmetic, as *interval programs*.

Appendix A describes a form of interval arithmetic for intervals whose endpoints are double-precision, IEEE-standard floating-point values. Appendix B gives an implementation of this interval arithmetic.

Aberth [Abe88] describes a version of interval arithmetic called *range arithmetic* that represents an interval as a midpoint and a distance, called a *range*, from this midpoint to the interval's endpoints. The operations of range arithmetic produce interval bounds that are not quite as tight as the bounds produced by conservative roundings to machine-representable endpoints, but the operations of range arithmetic are simpler and faster.

2.2 Elementary Properties

The great advantage of interval analysis is that it recognizes and bounds input, truncation and computational error, bounding every form of error except modeling error. Interval results definitely contain the desired quantities, and the lengths of the intervals clearly show the total uncertainty in these quantities. The results of scalar programs, by contrast, are typically close to the desired quantities, but not known to be higher, lower or exact, and nothing shows how uncertain they really are. Interval analysis pays for this great advantage with several disadvantages:

Interval arithmetic requires roughly twice as much calculation as scalar arithmetic, though the cost of this can be greatly reduced by performing many of the necessary calculations in parallel. The cost can also be reduced by using range arithmetic.

Interval arithmetic requires computing, in hardware or software, appropriate upward and downward rounding functions. IEEE-standard floating-point hardware includes these rounding functions, but they complicate it and all other representation systems that provide them. Range arithmetic, which produces slightly looser bounds, does not require these functions; Aberth's [Abe88] implementation of interval arithmetic runs on IBM machines where IEEE-standard floating-point is not available.

There are also significant problems with interval arithmetic that cannot be eliminated with more accurate representations of intervals' endpoints or more accurate operations on these endpoints. Interval arithmetic does not acknowledge, for instance, that input and computation errors are usually less than their extreme values and often cancel out. This is the main reason that input and calculation errors do not cause problems more often than they do. If n uniformly-distributed random error variables are added together, which corresponds to adding n intervals, the worst-case error is proportional to n , but for large n the error variance is roughly proportional to \sqrt{n} .

This defect is why our plan to consider alternative representation systems only as means for representing the endpoints of intervals was overly restrictive. Some of the representation systems described in Chapter 5, particularly the ones that use mediant rounding, are intended to exploit simplicity in computed results so that accumulated errors frequently cancel out.

Similarly, interval arithmetic does not reflect the relationships between errors in different computed values, so its computed bounds on these errors are often far too pessimistic. If a real number x is known only to lie in the interval $[-1, 1]$, for example, interval multiplication would say of x^2 only that it lies in $[-1, 1]$, when actually it must lie in $[0, 1]$. This problem cannot be solved by looking at individual operations and their arguments: If two real numbers x and y are both known only to lie in $[-1, 1]$, for example, but nothing is known about the relationship between x and y , then the product $x \cdot y$ is correctly only known to lie in $[-1, 1]$. This is the problem addressed by Matijasevich's ideas described in Section 2.5.

Operations other than the basic operations of interval arithmetic can sometimes be used to make deductions about relationships between the errors in different quantities. If a quantity is known to fall in two intervals, for example, it is known to fall in their intersection. Such operations are included in the interval arithmetic given in Appendix B and used in the Interval Newton's Method program discussed in Subsection 2.3.2 below.

2.3 Quantitative Results

This section describes quantitative results for a naive and a more sophisticated interval algorithm. The naive interval algorithm, which is a simple translation of a corresponding scalar algorithm, uses a Fast Fourier Transform (FFT) to multiply large integers. The more sophisticated interval algorithm uses an interval version of Newton's Method to find roots of polynomials. Results for the naive algorithm include results for both IEEE and VAX arithmetic; results for the more sophisticated algorithm use IEEE arithmetic only.

The IEEE-based interval computations were carried out with the interval arithmetic package given in Appendix B. All IEEE computations were carried out in double-precision floating-point on a Sun 3/60 with an MC8881 floating-point coprocessor, mask A95N, under release 3.5 of the Sun UNIX 4.2 operating system.

The VAX-based interval computations were carried out with interval arithmetic subroutines in the code given in Appendix C, Section 2. VAX

arithmetic [Dig81] does not have directed roundings, but for each operation produces a result that is either the representable value closest to the exact answer or the representable result with larger absolute value of the two equally-close representable values. VAX arithmetic is thus similar to IEEE arithmetic [IEE85] in its default round-to-nearest rounding mode, except that if there are two equally-near values VAX arithmetic takes the one with larger absolute value, while IEEE arithmetic takes the one whose last bit is 0. The VAX interval arithmetic subroutines compute absolute upper and lower bounds for intervals by adding or subtracting the quantity represented by the least significant bit in approximate upper and lower bounds computed with VAX arithmetic. All VAX computations were carried out in double-precision floating-point (D-floating) arithmetic on a VAX 11/750.

Every result is given in both ordinary scientific notation and in a "literal" form that is a string of hexadecimal digits specifying the exact bit pattern used to represent the result in whichever arithmetic system — either IEEE or VAX — is currently being used. For the interpretations of these strings, see [IEE85] for the IEEE values and [Dig81] for the VAX ones.

2.3.1 FFT Multiplication Results

We implemented scalar and interval versions of programs, both for IEEE and for VAX arithmetic, that use Fast Fourier Transforms (FFTs) to multiply large integers. The algorithm we implemented is described in Knuth [Knu81], Section 4.3.3, Part C. The interval versions of these programs were obtained by replacing floating-point values and operations by corresponding interval ones.

High accuracy is actually not necessary for this application, since the ideal final results are known to be integers and can thus be determined exactly from floating-point approximations that are in error by less than $1/2$. Since the ideal final results are known to be integers, though, it is possible to determine the error in the floating-point approximations easily, even though these approximations are only produced after a reasonably large amount of computation. Our programs were written to be run on integers that were small enough to be multiplied with machine hardware, and to produce output that was convenient for showing the errors in the floating-point results.

The interval version for IEEE arithmetic is given in Appendix C, Section 1, and uses the interval-arithmetic package given in Appendix B. The interval version for VAX arithmetic is given in Appendix C, Section 2. The two scalar versions of these programs were almost identical. The IEEE version is given in Appendix C, Section 3, and the VAX version was created by making the changes listed in comments in this code.

An edited version of the combined output from the four programs when they were used to multiply three particular pairs of integers is given in Appendix D. The editing consisted of removing redundant descriptive information and rearranging lines from the outputs to make it easier to compare them.

These FFT multiplication results support our interim report's comment, itself consistent with observations in the literature [SB80], that interval algorithms obtained by simply replacing operations on real numbers with the corresponding operations on intervals usually produce error bounds that are much too pessimistic. Even for the IEEE results, which are correctly rounded, the length of the interval calculated is sometimes over 2360 times the error in the corresponding scalar quantity. In three IEEE results, the scalar quantity is actually correct to all 52 of an IEEE double-precision value's bits of precision, while the corresponding interval reflects uncertainty in as many as 17 of the final bits.

The VAX interval results do not really reflect undue conservatism in interval bounds because the computed bounds are not optimal for the underlying floating-point arithmetic. These results do show, however, that the greater accuracy in VAX double-precision over IEEE double-precision arithmetic can produce more accurate results even without optimal rounding. This is discussed in Chapter 3.

2.3.2 Interval Newton's Method Results

We also implemented a more sophisticated interval algorithm, the Interval Newton's Method algorithm from Alefeld and Herzberger [Ale86]. The code for this algorithm is given in Appendix E, and uses the IEEE interval arithmetic package given in Appendix B.

This algorithm is based on the following observations. Define the length and midpoint functions on an interval $A = [a_0, a_1]$ by

$$\begin{aligned}\text{length}(A) &= a_1 - a_0 \quad \text{and} \\ \text{mid}(A) &= (a_0 + a_1)/2.\end{aligned}$$

Suppose f is a continuous function, and suppose the intervals $X^{(0)} = [x_0, x_1]$ and $M = [m_0, m_1]$ are such that $f(x_0) < 0$, $f(x_1) > 0$, and f has a root ξ in $X^{(0)}$ such that for all $x \in X^{(0)} - \{\xi\}$,

$$0 < m_0 \leq \frac{f(x) - f(\xi)}{x - \xi} = \frac{f(x)}{x - \xi} \leq m_1 < \infty.$$

Define intervals $X^{(k+1)}$ for all $k \geq 0$ inductively by

$$X^{(k+1)} = (\text{mid}(X^{(k)}) - \frac{f(\text{mid}(X^{(k)}))}{M}) \cap X^{(k)}.$$

In operations combining a real number and an interval, interpret the real number as a point interval. Then for all $k \geq 0$,

$$\begin{aligned}\xi &\in X^{(k)}, \\ X^{(0)} &\supset X^{(1)} \supset \dots X^{(k)}, \text{ and} \\ \text{length}(X^{(k+1)}) &\leq \frac{1}{2} \left(1 - \frac{m_0}{m_1}\right) \text{length}(X^{(k)}), \text{ so} \\ \lim_{k \rightarrow \infty} X^{(k)} &= \xi.\end{aligned}$$

The code in Appendix E acknowledges that the midpoint of an interval can usually only be computed approximately, but in evaluating f at an imprecisely-known midpoint it assumes that the possible range of values is contained in the interval determined by evaluating f , with rounding modes set appropriately, at the endpoints of the interval containing the midpoint. This assumption is reasonable, since the interval containing the midpoint will either be a point interval or have two consecutive representable values as its endpoints.

Samples of the output from this code are given in Appendix F. All of these results for the Interval Newton's Method are not only highly accurate, but perfect. As the results of Boehm's [Boe87] arbitrary-precision calculator and the "literal" results show, the intervals computed are the shortest possible machine-representable intervals containing the desired quantities.

2.3.3 Summary

These results show that a simple-minded application of interval analysis is unlikely to be useful, but that nontrivial interval algorithms can be surprisingly powerful and are worthy of further investigation. We did not expect the simple-minded interval algorithms, particularly those using the control of rounding available in IEEE arithmetic, to perform so poorly, and we did not expect the Interval Newton's Method algorithm to perform so well. This conclusion about the usefulness of simple-minded interval analysis is also supported by our experience, described in the next section, with trying to relate scalar and interval versions of asymptotic correctness.

2.4 Interval Asymptotic Correctness

We initially conjectured, since the results of interval calculations definitely contain desired exact values while the results of scalar ones are merely usually close to these exact values, that if a program could be proved asymptotically correct when the values of its variables and operations were reinterpreted as intervals and interval operations then the program would not only be asymptotically correct but *effectively* asymptotically correct — i.e., it would be theoretically possible to compute the degree of machine accuracy necessary to have the program produce a desired degree of output accuracy.

In an attempt to prove this conjecture, we began developing a generalization of the Reals project's asymptotic semantics [ORA87] that would allow the values of variables to be either nonstandard real numbers or intervals of nonstandard real numbers, and allow the arithmetic operations on these variables to denote either operations on nonstandard reals or on intervals of nonstandard reals. During this effort, however, we found a counterexample to the intent of our conjecture.

This section first gives an informal definition of the notion of asymptotic correctness and an informal description of the interval version of it we investigated. It then gives and explains the example we found showing that merely reinterpreting scalar programs as interval ones and proving them intervally asymptotically correct does not give useful information about the degree of machine accuracy necessary to obtain a desired degree of output accuracy.

2.4.1 Asymptotic Correctness Definitions

Asymptotic correctness can be loosely defined in standard mathematical terms as follows: A program is asymptotically correct if it always halts and its outputs become arbitrarily accurate as it is run on a sequence of machines whose sets of machine-representable numbers become progressively larger and whose machine arithmetic operations become progressively more accurate and progressively less vulnerable to overflow. On such a sequence of machines, arbitrary real-number inputs can thus eventually be approximated arbitrarily accurately, and arbitrary real-number calculations can eventually be carried out arbitrarily accurately and without exceptions[ORA87].

For the practical purpose of proving programs asymptotically correct, though, it is more convenient to define asymptotic correctness in terms of *nonstandard models of analysis*. With this definition, it is possible to formally prove programs asymptotically correct from axioms that are only slightly more complicated than typical axioms for the reals numbers and their usual operations and relations. The Reals project [ORA87] developed this approach as a means of eliminating the most common bugs in numerical software.

A nonstandard model of analysis is very similar to the real numbers with their usual arithmetic operations and equality and order relations, but it contains *infinitesimal* numbers other than 0 which are smaller in magnitude than any nonzero real number, and *infinite* numbers which are larger in magnitude than any real number. The real numbers, or *standard reals*, with their usual arithmetic operations and relations occur as a substructure of every nonstandard model of analysis. A nonstandard real is called *finite* if it is bounded by some standard real. Two nonstandard reals are *infinitesimally close* if their difference is an infinitesimal. See introductory textbooks on nonstandard analysis, e.g., Hurd [HL85], for more information about nonstandard models of analysis.

In the nonstandard model of analysis formulation of asymptotic correctness, every standard real is assumed to be infinitesimally close to a machine-representable value, every machine arithmetic operation on arguments that are finite, except division by an infinitesimal, is assumed to produce a result infinitesimally close to the result of the corresponding ideal operation, and overflow is assumed to never occur for any operation whose result is finite.

The number of different states a program can assume is also assumed to be bounded by a nonstandard integer, though this nonstandard integer can be infinite. A program is then asymptotically correct if always halts after a nonstandard integer number of steps and its results are then infinitesimally close to ideal values for the function or relation the program is specified to compute. A program is asymptotically correct by the standard definition if and only if it is asymptotically correct by the nonstandard definition [ORA87].

For a nonstandard analysis formulation of interval asymptotic correctness, assume that every standard real is contained in an interval of infinitesimal length with machine-representable endpoints, assume that every interval operation on intervals with finite endpoints, except division by an interval containing an infinitesimal, produces an interval whose endpoints differ only infinitesimally from the endpoints of the interval produced by the corresponding ideal interval operation, and assume that overflow never occurs on any interval operation that produces an interval whose endpoints are finite. A program is then intervally asymptotically correct if it halts after a nonstandard integer number of steps and its results are then intervals of infinitesimal length containing ideal values for the function or relation the program is specified to compute.

2.4.2 Conjecture Problem Example

The program given in Appendix G gives an example of the problem we found with relating the original scalar version of asymptotic correctness to the interval version just defined. This program computes π with a power series. Its variables `high` and `low` contain computed approximations to initial segments of the power series. If they were computed exactly, `high` and `low` would be definitely greater than and definitely less than π , respectively, the value of `high` would decrease monotonically, and the value of `low` would increase monotonically. The program runs until `low` is at least as large as `high`, or the newly-computed value of `high` is not strictly less than `high`'s previous value, or the newly-computed value of `low` is not strictly greater than `low`'s previous value.

With a formalization of IEEE arithmetic in a nonstandard model of analysis, this program is provably asymptotically correct. Such a proof shows

that, after a nonstandard (infinite) integer number of steps, either one of the variables `m`, `pow5` or `pow239` overflows to $+\infty$, or the accumulated errors in computation cause `low` to equal or exceed `high`. In all cases, since division by $+\infty$ gives 0 as its result and causes values of `high` and `low` to be equal or unchanged, the program's loop termination condition is met and the program halts with both `high` and `low` infinitesimally close to π .

Note that the possibilities that arise in the nonstandard proof of asymptotic correctness reflect possibilities on real machines. On most machines with IEEE arithmetic, for example, the variable `pow239` will overflow to $+\infty$ before the loop terminates, and that will cause computed values for `high` and `low` to be equal. On machines without IEEE arithmetic, the program has to be rewritten so that the loop terminates if either its termination condition is met or if overflow occurs.

If the program were reinterpreted to make the values of the variables intervals of nonstandard real numbers, and to make machine arithmetic operations the corresponding machine operations on these intervals, the program would terminate when the computed intervals for `high` and `low` intersected, when a newly-computed interval for `high` intersected the previously-computed interval for `high`, or when a newly-computed interval for `low` intersected the previously-computed interval for `low`. With the interval interpretation, the program could be proved to terminate after going through its loop a nonstandard (infinite) integer number of times, with both of the intervals assigned to the variables `high` and `low` being of infinitesimal length and both containing points infinitesimally close to π .

The critical problem with the proof of interval asymptotic correctness would be that it would not determine which, *if either*, of the two intervals assigned to the variables `high` and `low` actually contains π . If the program terminated because one or both of the computed intervals for `high` and `low` stopped strictly increasing or decreasing, then these two intervals might be separated by an infinitesimally long gap containing π . For the definition of "interval asymptotic correctness" given above, this interval version of the program would not be asymptotically correct.

There are, of course, programs to compute intervals containing π that could, in the asymptotic case, be proved to compute intervals of infinitesimal length containing π . It is also true that the program given in Appendix G is

effectively asymptotically correct. The example shows, though, that a correspondence between effective asymptotic correctness and interval asymptotic correctness cannot be given, as we had hoped, by a simple reinterpretation of values and operations. Programs that are intervally asymptotically correct are likely to contain interval operations such as union and intersection that do not correspond to any scalar operations.

2.5 Matijasevich's Method

Y. Matijasevich [Mat85] has suggested a method for calculating rigorous bounds on the uncertainties in computed outputs that arise from uncertainties in inputs and approximate calculations. This method, which is essentially an efficient way of computing numerical bounds on partial derivatives, addresses the problem that there are correlations between the errors in different computed quantities that make the bounds given by interval arithmetic far too conservative. His method has serious limitations, but is worthy of further study.

To obtain Matijasevich's idea in its simplest form, first assume that the program P_0 takes m values x_0, \dots, x_{m-1} as inputs, computes distinct values x_m, \dots, x_{n-1} as intermediate results, computes these intermediate results in straight-line code that uses only binary arithmetical operations, and returns $y = x_{n-1}$ as its final result. Assume that all programs are written in a C-like language, so $=$ denotes assignment, $+=$ denotes incrementing the quantity on the left by the quantity on the right, and $-=$ denotes decrementing the quantity on the left by the quantity on the right. To define notation, assume the lines of P_0 computing intermediate results are of the form

$$x_i = x_{g(i)} \star_i x_{h(i)};$$

for $m \leq i < n$, where $\star \in \{+, -, \cdot, /\}$. The functions g and h identify the left and right arguments, respectively, of the operations giving the intermediate results. For the moment, assume that all machine operations are exact, and ignore program constants.

Matijasevich points out that ordinary interval arithmetic is equivalent to replacing P_0 with a new program P_1 that takes not only the m values

x_0, \dots, x_{m-1} , but also m error-bound values e_0, \dots, e_{m-1} , as inputs, and computes for each $m \leq i < n$ an error bound e_i such that x_i varies by no more than e_i from its computed value if every x_j , $0 \leq j < m$ varies by no more than e_j from its input value. The program P_1 can be obtained from P_0 by inserting code to compute the necessary error bound before each line computing one of the intermediate results. If $\star_i \in \{+, -\}$, for example, the code inserted is

$$e_i = e_{g(i)} + e_{h(i)};$$

if $\star_i = \cdot$ the code inserted is

$$e_i = e_{g(i)} \cdot |x_{h(i)}| + e_{h(i)} \cdot |x_{g(i)}| + e_{h(i)} \cdot e_{g(i)};$$

and if $\star_i = /$ the code inserted is

$$\begin{aligned} &\text{if}(|x_{h(i)}| < e_{h(i)}) \\ &\quad e_i = +\infty; \\ &\text{else} \\ &\quad e_i = \frac{e_{g(i)} + e_{h(i)} \cdot |x_{g(i)} / x_{h(i)}|}{|x_{h(i)}| - e_{h(i)}}; \end{aligned}$$

(This presentation is slightly simpler than Matijasevich's and assumes the presence of the IEEE-arithmetic value $+\infty$.)

Matijasevich next gives an efficient method for calculating the partial derivatives $\partial y / \partial x_j$ for $0 \leq j < m$ at the point (x_0, \dots, x_{m-1}) and bounding them as each of the x_j varies by no more than e_j . To calculate the partial derivatives and bound them, form the program P_2 as follows: First append to P_1 the lines

$$\begin{aligned} z_0 &= 0; \\ d_0 &= 0; \\ &\dots \\ z_{n-2} &= 0; \\ d_{n-2} &= 0; \\ z_{n-1} &= 1; \\ d_{n-1} &= 0; \end{aligned}$$

Then as i varies from $n - 1$ downward to 0, if $\star_i = +$ append the lines,

$$z_{g(i)} += z_i;$$

$$d_{g(i)} += d_i;$$

$$z_{h(i)} += z_i;$$

$$d_{h(i)} += d_i;$$

if $\star_i = -$ append the lines

$$z_{g(i)} += z_i;$$

$$d_{g(i)} += d_i;$$

$$z_{h(i)} -= z_i;$$

$$z_{h(i)} += d_i;$$

if $\star_i = \cdot$ append the lines

$$z_{g(i)} += z_i \cdot x_{h(i)};$$

$$d_{g(i)} += |z_i| \cdot e_{h(i)} + d_i \cdot x_{h(i)} + d_i \cdot e_{h(i)};$$

$$z_{h(i)} += z_i \cdot x_{g(i)};$$

$$d_{h(i)} += |z_i| \cdot e_{g(i)} + d_i \cdot x_{g(i)} + d_i \cdot e_{g(i)};$$

and if $\star_i = /$ append the lines

$$z_{g(i)} += z_i / x_{h(i)};$$

$$d_{g(i)} += \frac{d_i + |z_i / x_{h(i)}| \cdot e_{h(i)}}{|x_{h(i)}| - e_{h(i)}};$$

$$z_{h(i)} -= z_i \cdot x_{g(i)} / x_{h(i)}^2;$$

$$d_{h(i)} += \frac{|z_i| \cdot e_{g(i)} + d_i \cdot |x_{g(i)}| + d_i \cdot e_{g(i)} + \frac{|z_i| \cdot |x_{g(i)}| \cdot (2 \cdot |x_{h(i)}| \cdot e_{h(i)} + e_{h(i)}^2)}{x_{h(i)}^2}}{(|x_{h(i)}| - e_{h(i)})^2};$$

The computed value of z_i can be roughly described as $\partial y / \partial x_i$ evaluated at the point (x_0, \dots, x_{m-1}) for y expressed in terms of the x 's that have been considered so far. When program P_2 terminates, $z_j = \partial y / \partial x_j$ evaluated at

(x_0, \dots, x_{m-1}) for all $0 \leq j < m$. To name the region of points "near" the point (x_0, \dots, x_{m-1}) , let

$$B = [x_0 - e_0, x_0 + e_0] \times \dots \times [x_{m-1} - e_{m-1}, x_{m-1} + e_{m-1}].$$

The values d_i are such that if $\partial y / \partial x_i$ is evaluated at different points in B , the value of $\partial y / \partial x_i$ never differs from z_i by more than d_i . It is then true that for every point $(x'_0, \dots, x'_{m-1}) \in B$,

$$\sum_{j=0}^{m-1} (-|z_j| - d_j) \cdot e_j \leq y(x'_0, \dots, x'_{m-1}) - y(x_0, \dots, x_{m-1}) \leq \sum_{j=0}^{m-1} (|z_j| + d_j) \cdot e_j.$$

Further, the error bound

$$e = \sum_{j=0}^{m-1} (|z_j| + d_j) \cdot e_j$$

is optimal in the sense that ratio of the largest difference between the value of y at (x_0, \dots, x_{m-1}) and at another point in B to e tends to 1 as all of the e_j tend to 0. The technique thus eliminates the excess conservatism in ordinary interval arithmetic.

To deal with error in machine operations, use directed roundings to compute conservative upper bounds in all the calculations of the e_i and d_i , and use interval arithmetic in all the computations of the x_i and z_i . Constants in programs can be treated as additional inputs; a constant x_i that is machine-representable can be dealt with more simply than the true inputs because e_i , d_i and z_i are all 0.

Matijasevich's technique requires a significant amount of additional computation, but this amount is only proportional to the length of the original program. The final program P_2 is only a constant multiple longer than the initial program P_0 . By contrast, computing the partial derivatives $\partial x_i / \partial x_j$ at (x_0, \dots, x_{m-1}) for each i and every $0 \leq j < m$, as was suggested by Hansen [Han75], takes time proportional to the original program's length times m . Computing the partial derivatives by working *backwards*, by looking at successive *decreasing* values of i , produces the necessary results much faster; that is why Matijasevich called his technique a "posteriori" version of interval analysis.

The most significant problem with Matijasevich's technique is that it is difficult to apply it to programs with loops. If a program ran on a machine that stored all its intermediate results and also stored a record of which operations it performed, one could carry out his technique to produce a new program computing the final error bound e as before, but the space needed to store this new program would no longer be a constant multiple of the space needed to store the original program.

Matijasevich claims that for some mathematical problems, particularly calculating determinants by putting matrices into triangular form, programs exist that compute the error bound e by a similar technique that require storage space on the order of that required by the original program. He does not give references, though, and suggests that even when such programs exist they cannot be produced by a mechanical transformation. His technique is worthy of further attention because a mechanizable means might be found for taking advantage of posteriori calculation of partial derivatives and bounds on these derivatives.

Chapter 3

Floating-Point Arithmetic

We noted in our interim report [ORA88] that one of the “alternatives” to floating-point arithmetic that should be considered was floating-point itself. We suggested that sufficiently precise versions of floating-point arithmetic, preferably ones meeting the IEEE standard, might suffice to make the asymptotic model realistic. We expected that such versions would have the single serious drawback that their operations would be slower, and expected that they could be implemented so that this slowing would be minimal.

We promised to investigate the costs and benefits of different versions of floating-point arithmetic, to perform a specific implementation to compare IEEE and VAX arithmetic, to investigate the theoretical speed limits of IEEE and other forms of floating-point arithmetic, and to look into whether alternate forms of floating-point arithmetic might facilitate parallel computation. This chapter comments on the realism of the asymptotic model, discusses the costs of precision in floating-point arithmetic, and gives empirical results comparing IEEE and VAX arithmetic. It also describes a version of floating-point arithmetic that facilitates parallel computation.

3.1 Realism of the Asymptotic Model

We have informally defined the notion of asymptotic correctness [ORA87] and its formulation in nonstandard models of analysis [HL85] in Section 2.4.

We take the *asymptotic model* of computation to be the one that assumes valid machine operations on finite quantities produce results that differ only infinitesimally from their ideal values. Although the Reals project developed the asymptotic model as an idealization that could be used to eliminate most bugs in numerical software, it is obviously a formalization of an "extremely precise" form of floating-point arithmetic, and is at least partly realistic for every accurate implementation of floating-point arithmetic.

As we noted in our interim report, a version of floating-point arithmetic with 256-bit mantissas and correctly rounded operations would be unlikely to accumulate computational errors large enough to be significant parts of measurable quantities in any calculation that could be carried out in a realistic time. In such a floating-point arithmetic, the computational errors do behave like the infinitesimals in nonstandard models of analysis. We set out to determine how much smaller the mantissas could be to give floating-point values that would still have this property.

We found, however, that there are situations in which no reasonable amount of precision in floating-point arithmetic suffices to make the asymptotic model accurate. The fragment of C code

```
x = 2.0;
for(i=0; i < 10000; ++i)
    x = sqrt(x);
for(i=0; i < 10000; ++i)
    x = x*x;
y = x;
```

for example, assigns *y* a value infinitesimally close to 2.0 in the asymptotic model, but on an accurate computer with fewer than thousands of bits in its floating-point values assigns *y* the value 1.0. Still, such examples are extremely unrealistic, so sufficiently precise floating-point values can be expected to make the asymptotic model realistic for most code.

The extended-precision values in the IEEE standard were chosen to have a range so large that "over/underflows, which mostly occur during intermediate calculations, would almost disappear", and to be precise enough that "for many calculations, rounding errors [are] really negligible" [KP79]. Ordinary double-precision values are also often considered to have this precision

property [Knu81]. This can be construed as saying that IEEE extended-precision values, or even ordinary double-precision values, are thought of as making the asymptotic model sufficiently accurate for most applications; the asymptotic model gives a precise interpretation of "really negligible".

3.2 Costs of Accuracy

Although we emphasized time as the critical cost in our interim report, the time needed to perform operations is not necessarily the most significant cost of using more precise floating-point values. The space needed to store these values was considered so expensive by those choosing the IEEE standard that extended-precision values were intended to be used only as intermediate values [KP79].

More importantly, even if fast algorithms theoretically exist for performing operations on larger floating-point numbers with little loss of speed (e.g., algorithms using Wallace [Wal64] or Dadda [Dad65] trees that perform multiplications in times proportional to the logarithm of the number of bits), the size of the necessary integrated circuits can be prohibitive. "Silicon real estate" is very expensive, so much of the literature on hardware design evaluates area/time tradeoffs [BPTP87, Fan87, HC87, PSG87, Sha87].

Further, fast algorithms can require more complicated integrated circuits that are more difficult to manufacture and have a lower yield of nondefective circuits than do slower algorithms. Simplicity is enough of an advantage in integrated circuit manufacture that there is engineering interest in theoretically slower algorithms with simpler hardware implementations [PSG87, Sha87].

Finally, although there are fast algorithms for addition, subtraction and multiplication that can be modified to produce correctly-rounded results consistently with the IEEE standard, the fastest algorithm ordinarily suggested for division, an iterative one [Knu81], does not produce correctly-rounded results and so cannot be modified to fit the IEEE standard (see e.g., [Fan87]).

The iterative division algorithm takes time that is proportional to $\log n$, where n is the number of bits in the mantissas of the numbers being divided. Although we did not locate proofs that division algorithms compatible with the IEEE standard which require only time proportional to $\log n$ do not

exist, that is suggested by the engineering literature, which contains descriptions of slower "fast" division algorithms compatible with the IEEE standard [BPTP87,Fan87,PSG87]. These algorithms are similar to ordinary long division but use redundant digit-sets (described briefly in Section 3.4 below) and nonrestoring division to simplify the multiplication of the divisor by the next digit in the result and to avoid having to "undo" a subtraction if the "guess" as to the next digit was too high [Atk68]. While these algorithms are much faster than other division algorithms, they still require time proportional to n .

The issue of whether the IEEE standard forces division algorithms to take times greater than constant multiples of the time required by the iterative algorithm did not arise in debates over adoption of this standard; see [Cod79, Fel79,FW79,KP79,PS79] and [Cod81,Coo81,Dem81,Hou81,TP81]. For practical purposes, large numbers of bits are not considered for operations that are to be implemented in hardware, particularly hardware constructed in accordance with the IEEE standard.

3.3 IEEE and VAX Arithmetic

We carried out the implementation of scalar and interval versions, for IEEE and VAX arithmetic, of programs to compute products with Fast Fourier Transforms. The code for these programs is in Appendices B and C, and an edited version of their output is in Appendix D. These programs and their output were described in Section 2.3. This section describes characteristics of their output that are significant for comparing IEEE and VAX arithmetic.

Comparing IEEE arithmetic and VAX arithmetic is essentially a matter of comparing more logical rounding with greater precision. In comparing roundings, the results of both IEEE and VAX arithmetic, for IEEE arithmetic in its default round-to-nearest mode, are typically produced as if these results were first computed exactly, then rounded to the nearest representable value. If an exact result is equally near two representable values, though, VAX arithmetic rounds it to the one with larger magnitude while IEEE arithmetic rounds it to the one whose least significant bit is 0. Rounding errors are thus more likely to accumulate in VAX than in IEEE arithmetic.

In comparing precisions, both VAX and IEEE double-precision floating-point values occupy 64 bits. On the VAX [Dig81], 55 of these bits code the mantissa and 8 of them code the exponent; in IEEE arithmetic [IEE85], 52 bits code the significand and 11 code the exponent. (The IEEE standard uses "significand" instead of "mantissa" because "significands" are defined for things such as infinite and "Not a Number" values for which "mantissas" are not defined.) In both VAX and IEEE double-precision values, the mantissa/significand is always normalized and its leading 1 is not explicitly recorded, so the two versions actually have 56 and 53 bits of precision, respectively. IEEE arithmetic thus sacrifices precision for moderately-sized numbers in order to handle numbers of more varying magnitudes.

The interval results show that even with a much cruder algorithm for determining interval endpoints the VAX results were better than the IEEE results in all except one case in which the IEEE result was not a point interval. (The VAX endpoint algorithm was too crude to give VAX arithmetic a chance to produce point intervals.) The scalar results were more evenly matched; in three of the eight cases in which both results were not perfect, the IEEE results were better. These results indicate that having control of rounding modes is not in itself worth three additional bits of precision, but that avoiding rounding errors that tend to accumulate is almost worth those three bits.

3.4 Floating-Point and Parallel Processing

We did find information in the engineering literature on alternative representations of floating-point values that facilitate performing many different operations in parallel. The basic idea, which is also used in the lexicographically-coded continued fraction version of finite rational arithmetic to be described in Section 5.2, is of doing *on-line* arithmetic.

In on-line arithmetic, each argument to an operation is represented as a sequence of generalized digits that could be bits, ordinary digits, negative digits, or so forth. The algorithm performing the operation inputs successive digits from its inputs and produces a sequence of similar successive digits as its output.

Some definitions of on-line arithmetic (e.g., [TE77]) also impose the additional restrictions that the values are represented so that their most significant digits occur first and that each operation produces the j th digit of its output after taking in at most $j + \delta$ digits of its inputs, where δ is a positive integer called the operation's *delay*. With the more restricted definition, on-line operations can perform variable-precision arithmetic and produce more or fewer result digits as desired; there is no problem of having a potentially infinite wait for the next digit.

With on-line arithmetic, performing many computations in parallel is simple, and no potentially-intractable problems with making sure the results of one computation are available before they need to be used in another one need to be solved. The expression

$$z = c_0 + x \cdot (c_1 + x \cdot (c_2 + x \cdot c_3)),$$

can be evaluated as

$$z = S_0(c_0, P_0(x, S_1(c_1, P_1(x, S_2(c_2, P_2(x, c_3)))))),$$

where S_0 , S_1 and S_2 denote circuits producing sums and P_0 , P_1 and P_2 denote circuits producing products. The digits of x can be duplicated and sent to P_0 , P_1 and P_2 simultaneously. As soon as P_2 begins to produce result digits, S_2 can begin combining them with digits from c_2 while P_2 continues taking in digits from x and c_3 . As soon as S_2 begins producing result digits P_1 can start operating while S_2 and P_2 continue operating, and as soon as P_1 begins producing result digits S_1 can begin operating while P_1 , S_2 and P_2 continue operating. Eventually, parts of all the necessary operations can be being performed simultaneously. The more complicated evaluating an expression becomes, the greater is the possibility for performing many operations in parallel.

Ercegovac and Watanuki [WE81] have developed an on-line version, using the more restricted definition of "on-line", of floating-point arithmetic. Their version of floating-point uses a *maximally redundant* set of digits to code the mantissas. For numbers to a base b , the maximally redundant set of digits is

$$\{-(b-1), \dots, -1, 0, 1, \dots, (b-1)\}.$$

As notation for negative digits, let the negative of an ordinary digit be given by putting a bar over the digit, so $\bar{3} = -3$, $\bar{9} = -9$, and so on. The nonzero

numbers with redundant digit-sets are otherwise interpreted as if they were ordinary base- b floating-point numbers whose mantissas have magnitudes between 0 and 1. For $b = 10$, for example, the value with mantissa $1\bar{2}5$ and exponent 2 denotes the number $1 \cdot 10 + (-2) \cdot 1 + 5 \cdot 0.1 = 8.5$.

Ercegovic and Watanuki's system treats the exponent of a floating-point value as a single digit, and each operation determines the exponent and first digit of its result after reading the exponents and first digits of its arguments. Their on-line floating-point arithmetic operations are then essentially given by corresponding on-line arithmetic operations for the mantissas. Addition and subtraction are easy — we will briefly describe base-10 addition to show how the redundant digit set is used — and multiplication and division algorithms are given in [TE77].

To add, take the exponent of the result to be the larger of the exponents of the arguments, and shift the mantissa of the argument with the smaller exponent to the right, filling in a leading 0 and incrementing that argument's exponent with each shift, until the exponents are equal. Next output the successive digits of the result's mantissa such that:

1. The first j (or $j + 1$ if there is an initial carry) digits of the result's mantissa equal the sum of the first j digits of the properly-aligned arguments' mantissas; and
2. The last (the j th or $j + 1$ st) digit of the result's mantissa is not 9 or $\bar{9}$.

The condition about the last digit can be achieved since $9 = 1\bar{1}$ and $\bar{9} = \bar{1}1$, and any carry cannot affect more than the next-to-last digit — by induction the next-to-last digit was not 9 or $\bar{9}$. It is thus possible to add the mantissas with a delay of at most 1.

Redundant digit sets essentially get their power because it is not necessary to determine each successive result digit exactly. It is sufficient to output a result digit that is close to the value it would have in a nonredundant system, then correct it later if necessary. In ordinary base-10, for example, the third digit of a number whose partial computation begins $0.32999\dots$ is uncertain, but in redundant base-10 it can be taken to be 3; if the number turns out to be smaller, say 0.329998 , express it as $0.33000\bar{2}$.

Chapter 4

Mathematics for Alternatives

This chapter describes approximate rational arithmetic, mediant rounding, standard and generalized continued fractions and Gosper's algorithm. These concepts are used in several of the alternative representation systems from the literature that are described in Chapter 5, particularly the *fixed-slash*, *floating-slash* and *lexicographic continued fraction (LCF)* representations by Matula and Kornerup [MK80,KM81,MK83,KM83a,MK85,KM85,KM88].

4.1 Approximate Rational Arithmetic

Our interim report [ORA88] only discussed rational arithmetic as a means for doing exact computation. It only considered rationals given as pairs of lists specifying arbitrarily-large integers, or as scaled versions of such rationals multiplied by powers of a fixed base. We argued that the inability of such representations to naturally discard information, with the loss of speed and excess use of memory that this implies, makes rational arithmetic impractical for typical applications.

There are alternatives, though, of *approximate rational arithmetic*. These define approximate arithmetic operations on finite sets of rationals, rationals that can each be stored in a fixed number of bits. In these arithmetics, the results of an operation are the rationals that would be obtained by applying a fixed *rounding*, as in Section 2.1, to the operation's ideal results, rounding

them to representable values. These arithmetics avoid, to differing degrees, the problem of a steadily-growing use of time and space that is implied by exact rational arithmetic, but still allow many rationals to be represented exactly and many operations on rationals to be performed exactly.

With this very general definition, some versions of floating-point arithmetic, particularly the VAX and IEEE ones, are approximate rational arithmetics; floating-point values are rationals, and these arithmetics effectively obtain their results by applying fixed rounding functions. All versions of what are normally called approximate rational arithmetics, though, including all those proposed by Matula and Kornerup, differ from floating-point in exactly representing large numbers of simple rationals, e.g., $1/3$. Further, in all these arithmetics $-x$ and $1/x$ are representable whenever x is; $1/0$ is representable and is taken as $+\infty$.

In typical floating-point calculations, even input floating-point values are thought of as approximations to ideal real numbers that are usually irrational. We will call the numbers arising in a calculation *rational* if the ideal inputs, exact intermediate results, and exact outputs of the calculation are rational, and call these numbers *irrational* otherwise. The general hope behind approximate rational arithmetics is that representable rationals will occur frequently in calculations, so that the approximate arithmetics will often capture the advantages of exact rational arithmetic while still maintaining adequate precision in other cases.

All versions of approximate rational arithmetic provide means for discarding information, particularly in the process of replacing exact results by their representable approximations. The different approximate rational arithmetics vary in which rationals are representable, which rounding implicitly computes representable approximations, and which exceptions, error indications and error values are returned in exceptional cases. Choosing among such systems requires making trade-offs between the systems' time and space requirements, the range of the numbers they represent, and the utility of any additional information they provide.

Call a fraction p/q *simpler* than a fraction p'/q' if $p \leq p'$, $q \leq q'$, and at least one of these inequalities is strict. Let a *simple chain* be a finite set of irreducible fractions, ordered by the usual order on the real numbers, with the property that all irreducible fractions simpler than some member of the

chain are also in the chain. Every simple chain contains the rationals $0/1$ and $1/0$, which represent 0 and $+\infty$ respectively; call a simple chain *trivial* if it contains only these rationals. In each of the approximate rational arithmetics considered in Chapter 5, 0 is representable, the negative of every representable value is representable, and the nonnegative representable rationals form a simple chain.

As an example, the set

$$\left\{ \frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1}, \frac{3}{2}, \frac{2}{1}, \frac{3}{1}, \frac{1}{0} \right\}$$

is a simple chain, but would not be if $3/1$ were removed; $3/1$ is simpler than $3/2$, and $3/2$ is in the set. As the example indicates, the members of a nontrivial simple chain are not evenly spaced.

The different arithmetics proposed by Matula and Kornerup listed above all use a process, described in the next section, called *mediant rounding* to round unrepresentable reals to representable ones. The approximate arithmetic by Hwang and Chang [HC78], and the one by Yoshida [Yos83], do not; they use roundings similar to the roundings for VAX arithmetic and for IEEE arithmetic in its default, round-to-nearest mode.

Mediant rounding quite often does not round unrepresentable reals to the nearest representable ones. This rounding, based on the theory of best rational approximations, is biased in favor of simplicity, so it is more likely to give simple rationals than complicated ones as results.

With mediant rounding, errors in intermediate results often cancel out completely in computations on rational numbers. Errors in computations whose ideal inputs are irrational, though, tend to be larger with mediant rounding than they would be with round-to-nearest rounding. Empirical results to these effects are described in Subsection 5.1.4.

The uneven spacing of members of simple chains, and the fact that mediant rounding leads errors to cancel out, both have as a consequence that our interim-report [ORA88] intention to view alternative representation systems only as means for representing the endpoints of intervals was too restrictive. None of the fixed-slash, floating-slash or LCF representations are appropriate for representing the endpoints of intervals because the degree of conservatism introduced by bounding reals with representable values varies greatly

— measures on this variability are given in Subsections 5.1.3 and 5.2.2. The characteristic of mediant rounding that it tends to make errors cancel out gives these representations a significant advantage over other representations, though, an advantage contrasting with one of interval arithmetic's greatest weaknesses — c.f., Section 2.3.

Still, the importance of mediant rounding's ability to make errors cancel out should not be exaggerated. Errors tend to largely cancel out even without roundings biased in favor of the correct results. Note in Appendix D that, even when the IEEE interval results indicate some uncertainty, three of the eight scalar FFT-multiply results for IEEE arithmetic, and one of the eight for VAX arithmetic, are exactly correct.

4.2 Mediant Rounding

This section defines mediant rounding and gives some of its properties. The following definitions and results are classic pieces of number theory [HW60]; proofs of the results are repeated in [MK80].

Let the letters p , q , and their primed variants all denote nonnegative integers. Call two fractions p/q and p'/q' *adjacent* if $|p/q - p'/q'| = 1/qq'$, or equivalently if $|pq' - p'q| = 1$. Note that adjacent fractions must be irreducible, and that except for the pair $(0/1, 1/0)$ one of the two must be simpler than the other.

Let the *mediant* of p/q and p'/q' be $(p + p')/(q + q')$. If p/q and p'/q' are adjacent and $p/q < p'/q'$, then the following are all true:

1. $p/q < (p + p')/(q + q') < p'/q'$;
2. p/q and p'/q' are both simpler than $(p + p')/(q + q')$; and
3. $(p + p')/(q + q')$ is the simplest rational strictly between p/q and p'/q' .

Further, consecutive members of any simple chain are adjacent. (No, this is not obvious, and neither is the third of the facts just listed.)

From these facts it follows that if p/q and p'/q' are consecutive members of any nontrivial simple chain and $p/q < p'/q'$, then

1. $p'/q' - p/q = 1/qq'$;
2. $(p + p')/(q + q')$ is not as simple as either of p/q and p'/q' , and does not belong to the simple chain; and
3. $(p + p')/(q + q')$ is farther from the simpler of p/q and p'/q' .

If R , a set of representable rationals, consists of the members of a simple chain and their negations, define the *mediant rounding function* Φ_R from arbitrary reals to members of R as follows [MK80]: Let x be a real number. If $x \in R$, let $\Phi_R(x) = x$. If $x < 0$ let $\Phi_R(x) = -\Phi_R(-x)$. If x is positive and not in R , there exist two consecutive fractions p/q and p'/q' in R such that $p/q < x < p'/q'$. In this case, let $m = (p + p')/(q + q')$ be the mediant of p/q and p'/q' , then let $\Phi_R(x) = p/q$ if $x < m$, let $\Phi_R(x) = p'/q'$ if $x > m$, and let $\Phi_R(x)$ be the simpler of p/q and p'/q' if $x = m$.

Since the mediant $(p + p')/(q + q')$ of adjacent fractions p/q and p'/q' is farther from the simpler of the two, mediant rounding rounds most of the reals in the interval from p/q to p'/q' to the simpler of the two. Further, the interval of reals that are rounded to p/q is longer the simpler p/q is. These phenomena give the precise meaning of the statement that mediant rounding is “biased in favor of simplicity”.

Actually performing mediant rounding uses the theory of continued fractions, which is also basic to the LCF representation. We will give an explicit algorithm for computing Φ_R in the next section, a section that defines standard continued fractions, defines concepts and notations associated with them, and states some of their properties.

4.3 Standard Continued Fractions

For any real x , there exist unique reals n_0 and r_0 such that n_0 is an integer, $0 \leq r_0 < 1$, and

$$x = n_0 + r_0.$$

If $r_0 \neq 0$, expressing r_0 as $1/(1/r_0)$ and repeating the process gives unique reals n_1 and r_1 such that n_1 is an integer, $0 \leq r_1 < 1$, and

$$x = n_0 + \frac{1}{n_1 + r_1}.$$

Similarly, if $r_1 \neq 0$ there exist unique reals n_2 and r_2 such that n_2 is an integer, $0 \leq r_2 < 1$, and

$$x = n_0 + \frac{1}{n_1 + \frac{1}{n_2 + r_2}}.$$

The sequence of integers n_0, n_1, \dots generated in this way is called the *standard continued fraction* for x , and the successive integers are called the fraction's *partial quotients*. For the remainder of this section, assume that all continued fractions are standard.

It is traditional to write a sequence of integers to be interpreted as the partial quotients of a continued fraction in brackets, so for x and the n_i as above,

$$x = [n_0, n_1, \dots].$$

Any rational's continued fraction is finite — i.e., for x and the n_i and r_i as above, if x is rational the process eventually terminates for some i with $r_i = 0$. As an example,

$$2.31 = [2, 3, 4, 2, 3].$$

Since for any integer n , $n = (n - 1) + 1/1$, relaxing the condition $r_i < 1$ to $r_i \leq 1$ gives every rational two different continued fractions; e.g.,

$$2.31 = [2, 3, 4, 2, 3] = [2, 3, 4, 2, 2, 1].$$

The usual definition of continued fractions, with its $r_i < 1$ restriction, eliminates the second of these two possibilities. In the LCF representation, however, every finite continued fraction must have an even number of partial quotients, so this representation eliminates the first of the two possibilities for 2.31.

Many important irrational numbers have continued fractions that are infinite but particularly simple. It is true, for example, that

$$\sqrt{2} = [1, 2, 2, 2, 2, \dots]$$

and that

$$e = [2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, \dots].$$

(Here e is the base of the natural logarithms.) Continued fractions do not depend on the choice of a base, so continued fraction representations do not

depend on properties of the numbers 10, 2, 8 or 16 as decimal, binary, octal or hexadecimal representations do.

For a real $x = [n_0, n_1, \dots]$, define the sequences of integers $\langle p_i \rangle$ and $\langle q_i \rangle$ by

$$\begin{aligned} p_{-2} &= 0, \\ q_{-2} &= 1, \\ p_{-1} &= 1, \\ q_{-1} &= 0, \\ p_i &= n_i \cdot p_{i-1} + p_{i-2}, \text{ and} \\ q_i &= n_i \cdot q_{i-1} + q_{i-2} \end{aligned}$$

for all $i \geq 0$. The rationals p_i/q_i are called the *convergents* of x . For all $i \geq 0$, the integers p_i and q_i and the convergents of x have the following properties, all of which are classic number theory results [HW60] repeated in [MK80]:

1. $p_i/q_i = [n_0, n_1, \dots, n_i]$;
2. $\gcd(p_i, q_i) = 1$;
3. (Adjacency) $q_i p_{i-1} - p_i q_{i-1} = (-1)^i$;
4. (Alternating convergence)

$$\frac{p_0}{q_0} < \frac{p_2}{q_2} < \dots < \frac{p_{2j}}{q_{2j}} < x < \dots < \frac{p_{2j-1}}{q_{2j-1}} < \dots < \frac{p_3}{q_3} < \frac{p_1}{q_1};$$

5. (Best rational approximation) If $r/s \neq p_i/q_i$ and $s \leq q_i$ then

$$\left| \frac{r}{s} - x \right| > \left| \frac{p_i}{q_i} - x \right|;$$

6. (Quadratic convergence)

$$\frac{1}{q_i(q_{i+1} + q_i)} < \left| \frac{p_i}{q_i} - x \right| \leq \frac{1}{q_i q_{i+1}}.$$

We can now give an explicit algorithm for computing the mediant rounding function Φ_R , an algorithm strongly related to Euclid's greatest common denominator algorithm [KM83a]. For an arbitrary real number x and a set of

rational numbers R consisting of the members of a simple chain and their negations, compute the convergents $p_0/q_0, p_1/q_1, \dots$ of $|x|$. Then

$$\Phi_R(x) = \begin{cases} x & \text{if } x \in R, \\ -\Phi_R(-x) & \text{if } x < 0, \\ p_i/q_i & \text{if } x > 0, p_i/q_i \in R, p_{i+1}/q_{i+1} \notin R. \end{cases}$$

As a point of interest, while the last convergent of x that is in R gives the mediant-round of x to a value in R , a method similar to the one for computing successive convergents from partial quotients gives values between the last convergent in R and the first one not in R , and it can be used to find the member of R that is actually closest to x [Lov86].

4.4 Generalized Continued Fractions

A more general form of continued fractions gives a faster algorithm for doing mediant rounding, a possible extension of the LCF representation, and a variant of some of the constructive-real representations to be considered in Chapter 6. These generalized continued fractions give redundant representations of the reals that have the same sorts of advantages that redundant-digit-set representations do. The basic idea of generalized continued fractions is to stop considering only integers *less than* or equal to particular reals.

For any real x , let m_0 be any integer such that $|x - m_0| < 1$, and let s_0 be the real such that

$$x = m_0 + s_0.$$

If $s_0 \neq 0$, express s_0 as $1/(1/s_0)$ and repeat the process, finding an integer m_1 and a real s_1 such that $|s_1| < 1$ and

$$x = m_0 + \frac{1}{m_1 + s_1}.$$

Similarly, if $s_1 \neq 0$, find an integer m_2 and a real s_2 such that $|s_2| < 1$ and

$$x = m_0 + \frac{1}{m_1 + \frac{1}{m_2 + s_2}}.$$

Call any sequence of integers m_0, m_1, \dots generated by this process a *generalized continued fraction* for x . As before, call the successive integers the fraction's *partial quotients*.

Unlike standard continued fractions, many different generalized continued fractions can represent the same real, even if a convention is adopted, say, to force the number of partial quotients in a generalized continued fraction to be even. However, convergents can be defined for generalized continued fractions just as before, and they have many of the same properties.

If generalized continued fractions are restricted so that their partial quotients are "best possible" integer approximations, the restricted generalized continued fraction for every real is unique. Using the notation just given, let the *optimal* (generalized) continued fraction for x be the one such that for all i , $|s_i| \leq 1/2$ and s_i has the same sign as m_i if $|s_i| = 1/2$. The optimal continued fraction for e , for example, is given by

$$e = [3, -4, 2, 5, -2, -7, 2, 9, -2, -11, 2, 13, -2, -15, \dots].$$

A standard continued fraction that does not have 1 as one of its partial quotients is optimal, though 1 can (rarely) occur as a partial quotient in a continued fraction that is optimal. All except possibly the first of the partial quotients of an optimal continued fraction have magnitude at least 2. One can show that the sequence of convergents for a real's optimal continued fraction is a subsequence of the sequence of convergents for the real's standard continued fraction. (C.f. [KM83a].) The convergents of a generalized, particularly optimal, continued fraction can thus converge to a real more quickly than do the convergents of that real's standard continued fraction.

As an example, the number $49/30$ has the standard continued fraction

$$49/30 = [1, 1, 1, 1, 2, 1, 2],$$

with convergents $1/1, 2/1, 3/2, 5/3, 13/8, 18/11$, and $49/30$. The same number has the optimal continued fraction

$$49/30 = [2, -3, 4, -3],$$

with convergents $2/1, 5/3, 18/11$ and $49/30$.

Matula and Kornerup give an algorithm in [KM83a] for performing mediant rounding that computes generalized continued fractions and always *attempts* to impose the restriction $|s_i| \leq 1/2$. This algorithm succeeds in imposing this restriction most of the time, so it determines “almost optimal” generalized continued fractions. An “almost optimal” generalized continued fraction’s sequence of convergents is not necessarily a subsequence of the corresponding standard continued fraction’s convergents, but does determine the last representable member of the standard continued fraction’s sequence of convergents [KM83a], and hence can be used to do mediant rounding. This algorithm is faster than the algorithm given earlier because “almost optimal” continued fractions’ convergents typically converge faster than the corresponding standard continued fractions’ convergents do.

In the terminology introduced above, finding the optimal continued fraction for an x being determined by successive approximations is only difficult when one of the $|s_i| \approx 1/2$. In practical algorithms where this problem arises, like the faster mediant rounding algorithm or the generalizations of Gosper’s algorithm to be discussed in Section 4.5 and in Chapter 6, the algorithm makes a simple trade-off between the benefit of having optimal continued fractions rather than nearly-optimal ones and the cost of distinguishing optimal from nearly-optimal ones. Some of the algorithms in Chapter 6 would not work without this flexibility, which arises from the redundancy in generalized continued fractions. The redundancy in generalized continued fractions also provides a possible means of extending the LCF representation discussed in Section 5.2.

4.5 Gosper’s Algorithm

The LCF representation, some results in Chapter 6, and some of the research questions in Chapter 7, all use or refer to Gosper’s algorithm [Gos72] for finding the sum, difference, product or quotient of two (standard or generalized) continued fractions. We will first describe the algorithm for standard continued fractions, then indicate how to extend it to do arithmetic on LCF encodings, and finally tell how to adapt it to generalized continued fractions.

First assume that all continued fractions are standard, and identify reals and their continued fractions. Given continued fractions for the reals x and y ,

Gosper's algorithm computes the successive partial quotients in the continued fraction for z , where

$$z(x, y) = \frac{axy + bx + cy + d}{exy + fx + gy + h}$$

for almost any integers a, b, c, d, e, f, g and h . The only restrictions on a through h arise because of problems with possibly dividing by 0.

The algorithm treats a through h as assignable variables whose values can be changed, and operates by inputting partial quotients from x and y , outputting partial quotients to z , and making corresponding changes to the variables a through h . The algorithm also treats x , y and z as assignable variables whose values can change. To define notation, if

$$x = [n_0, n_1, n_2, \dots], y = [m_0, m_1, m_2, \dots] \text{ and } z = [k_0, k_1, k_2, \dots],$$

let

$$x' = [n_1, n_2, \dots], y' = [m_1, m_2, \dots], \text{ and } z' = [k_1, k_2, \dots].$$

Changing the sequence of remaining partial quotients for x from (n_0, n_1, \dots) to (n_1, \dots) , which the algorithm often does, is equivalent to replacing the value of assignable variable x with the value x' , and similarly with changing the sequences of remaining partial quotients for y or z .

Gosper's algorithm inputs the first partial quotient of one of its inputs, say n_0 of x , and updates the integers a through h as follows:

$$\begin{array}{ll} a & \longrightarrow a \cdot n_0 + c, \\ b & \longrightarrow b \cdot n_0 + d, \\ c & \longrightarrow a, \\ d & \longrightarrow b, \\ e & \longrightarrow e \cdot n_0 + g, \\ f & \longrightarrow f \cdot n_0 + h, \\ g & \longrightarrow e, \quad \text{and} \\ h & \longrightarrow f. \end{array}$$

By the relationship $x = n_0 + 1/x'$, the old value of $z(x, y)$ is then the new value of $z(x', y)$. When it takes a partial quotient from the continued fraction of x , though, the algorithm implicitly changes the (new) value of x to the (old)

value of x' , so the old value of $z(x, y)$ is equal to the new value of $z(x, y)$. This process is called *ingesting* a partial quotient from x . The process of ingesting a partial quotient from y is similar; the necessary updates to a through h can be calculated using the relationship $y = m_0 + 1/y'$.

After one or more of its initial partial quotients have been ingested, the value of x must be in the interval $[1, \infty)$, and similarly for y . If the denominator of the updated expression defining z cannot be 0, which it cannot be if the initial values of a through h defined one of the four operations of addition, subtraction, multiplication and division, $z(1, 1)$, $z(1, \infty)$, $z(\infty, 1)$ and $z(\infty, \infty)$ bound the possible values of $z(x, y)$ [KM88]. The values of z at these four extremes converge toward each other as the algorithm ingests more partial quotients from x and y . In general, ingesting a partial quotient of x reduces the uncertainty in z caused by uncertainty in x , and ingesting a partial quotient of y reduces the uncertainty in z caused by uncertainty in y .

It might be possible to increase the parallelism in computations by choosing the integers a through h to compute two or more operations at the same time. In that case, having the denominator of z be 0 can be a problem [KM88].

If a special symbol is used as an "end marker" for continued fractions, so that it is possible to detect when all the partial quotients of an input have been ingested, that input is called *exhausted*. It is only necessary to check two extremes to determine the possible range for $z(x, y)$ if one of x or y is exhausted, and the value of z is completely determined if both x and y are exhausted.

After Gosper's algorithm has ingested a large enough number of partial quotients from x and y , the integer portion of z , the first partial quotient of z 's continued fraction, is determined *whatever* the current values for x and y are. If this partial quotient is k_0 , the algorithm outputs k_0 and updates a through h as follows:

$$\begin{aligned}
a &\longrightarrow e, \\
b &\longrightarrow f, \\
c &\longrightarrow g, \\
d &\longrightarrow h, \\
e &\longrightarrow a - k_0 \cdot e, \\
f &\longrightarrow b - k_0 \cdot f, \\
g &\longrightarrow c - k_0 \cdot g, \text{ and} \\
h &\longrightarrow d - k_0 \cdot h.
\end{aligned}$$

By the relationship $z = k_0 + 1/z'$, the old value of $z'(x, y)$ is then the new value of $z(x, y)$. When it outputs a partial quotient of the continued fraction for z , though, the algorithm implicitly changes the (new) value of z to the (old) value of z' , so the continued fraction for the new value of z is the remainder of the continued fraction for the old value of z . This process is called *outputting* a partial quotient of z . If it is not yet possible to output the next partial quotient of z , a good strategy for the algorithm is to ingest a partial quotient from whichever of x and y seems to cause the most change in z as z varies between its extremes.

Following Matula and Kornerup [KM88], call the 8-tuple of integers a through h the *coefficient cube*; each number corresponds to a corner of the cube. The processes of ingesting partial quotients of x and y and of outputting partial quotients of z thus cause changes in the coefficient cube, so the coefficient cube reflects the current status of the computation of z .

Matula and Kornerup [KM88] give a method for evaluating z at its extremes that reduces the amount of computation needed and also makes their extension of Gosper's algorithm to LCF encodings possible. They define an 8-tuple of integers A through H called a *decision cube* having the property that the four extreme values of z are given by

$$z(1, 1) = \frac{D}{H}, \quad z(\infty, 1) = \frac{B}{F}, \quad z(1, \infty) = \frac{C}{G}, \quad z(\infty, \infty) = \frac{A}{E}.$$

The initial entries of the decision cube can be computed from the initial entries of the coefficient cube by

$$\begin{aligned}
A &= a, & B &= a + b, & C &= a + c, & D &= a + b + c + d, \\
E &= e, & F &= e + f, & G &= e + g, & H &= e + f + g + h.
\end{aligned}$$

We will show how the algorithm updates the members of the decision cube as it ingests and outputs partial quotients after we give Matula and Kornerup's critical observation that the processes of updating the coefficient and decision cubes can be described as matrix multiplications.

Matula and Kornerup [KM88] note that the transformation in the coefficient cube produced by ingesting the partial quotient n_0 of x is equivalent to multiplying each of the two matrices

$$\begin{bmatrix} d & b \\ c & a \end{bmatrix} \text{ and } \begin{bmatrix} h & f \\ g & e \end{bmatrix}$$

by the matrix

$$\begin{bmatrix} 0 & 1 \\ 1 & n_0 \end{bmatrix}.$$

Similarly, the transformation in the coefficient cube produced by ingesting the partial quotient m_0 of y is equivalent to multiplying each of the matrices

$$\begin{bmatrix} d & c \\ b & a \end{bmatrix} \text{ and } \begin{bmatrix} h & g \\ f & e \end{bmatrix}$$

by the matrix

$$\begin{bmatrix} 0 & 1 \\ 1 & m_0 \end{bmatrix}.$$

Likewise, the transformation in the coefficient cube produced by outputting the partial quotient k_0 of z is equivalent to multiplying each of the matrices

$$\begin{bmatrix} d & h \\ b & f \end{bmatrix} \text{ and } \begin{bmatrix} c & g \\ a & e \end{bmatrix}$$

by the matrix

$$\begin{bmatrix} 0 & 1 \\ 1 & -k_0 \end{bmatrix}.$$

The corresponding transformations on the decision cube are given for the same ingesting of n_0 from x by multiplying the matrices

$$\begin{bmatrix} D & B \\ C & A \end{bmatrix} \text{ and } \begin{bmatrix} H & F \\ G & E \end{bmatrix}$$

by the matrix

$$\begin{bmatrix} 1 & 1 \\ n_0 & n_0 - 1 \end{bmatrix};$$

they are given for the same ingesting of m_0 from y by multiplying the matrices

$$\begin{bmatrix} D & C \\ B & A \end{bmatrix} \text{ and } \begin{bmatrix} H & G \\ F & E \end{bmatrix}$$

by the matrix

$$\begin{bmatrix} 1 & 1 \\ m_0 & m_0 - 1 \end{bmatrix};$$

and given for the same output of k_0 to z by multiplying the matrices

$$\begin{bmatrix} D & H \\ B & F \end{bmatrix} \text{ and } \begin{bmatrix} C & G \\ A & E \end{bmatrix}$$

by the matrix

$$\begin{bmatrix} 0 & 1 \\ 1 & -k_0 \end{bmatrix}.$$

The observation that the updates to the coefficient and decision cubes can be made by multiplying by matrices is significant because each of the matrices

$$\begin{bmatrix} 0 & 1 \\ 1 & n_0 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & m_0 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ n_0 & n_0 - 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ m_0 & m_0 - 1 \end{bmatrix}, \text{ and } \begin{bmatrix} 0 & 1 \\ 1 & -k_0 \end{bmatrix}$$

can be factored, making it possible to ingest or output "pieces" of partial quotients. This is critical to doing arithmetic in the LCF representation, which is described in Subsection 5.2.3.

The most serious "catch" in Gosper's algorithm, ignoring for the moment the possibility of ingesting or outputting "pieces" of partial quotients, is that the four possible extreme values of z might all be close to a particular integer k , with some slightly above k and some slightly below it. In such a situation, it is impossible to determine whether the next partial quotient of z is k or $k - 1$, and this situation might persist indefinitely as the algorithm ingests more and more partial quotients from x and y , even though the four possible extreme values of z would get closer and closer to k . Such a situation is

exactly like not being able to determine the third digit of an ordinary base-10 number whose partial, approximate computation begins 0.32999

Gosper's algorithm can be extended to generalized continued fractions. For x, x', y, y', z and z' as defined before, but without the restriction that all except the first of the partial quotients of x, y and z must be positive, the relationships $x = n_0 + 1/x', y = m_0 + 1/y'$ and $z = k_0 + 1/z'$ are still valid. Gosper's algorithm can thus be applied even if x, y and z are all generalized continued fractions. The only additional complications, described in Subsection 6.4.2, are that there are more possibilities for having 0 as the denominator of the expression defining z , and more extreme values to check in bounding z . Even after ingesting one or more partial quotients from a generalized continued fraction for x , for example, x might be any member of the set $(1, +\infty) \cup (-\infty, -1)$. Even if the generalized continued fraction for x is known to be optimal, after ingesting one or more partial quotients x can still be any member of the set $[2, +\infty) \cup (-\infty, -2]$.

Generalized continued fractions avoid the main "catch" in Gosper's algorithm. If there is an integer k such that all the possible values of z are less than distance 1 from k , then k can be taken as the next partial quotient of z . If all the possible values of z are less than distance $1/2$ from k , this choice of k is optimal. It might sometimes happen that it is impossible to find the *optimal* next partial quotient for z — in situations where the possible values of z cluster around a point midway between two consecutive integers — but it is always possible to find an *acceptable* next partial quotient for z . Deciding how many partial quotients to ingest in such a situation is just a matter of convenience, balancing the benefits of having optimal partial quotients against the costs of determining them.

Being able to ingest and output "pieces" of partial quotients lessens the impact of the main "catch" in Gosper's algorithm, but it does not eliminate it. We discuss this problem in Subsection 5.2.4, sketch the method Matula and Kornerup propose for solving it, and suggest that using extensions of Gosper's algorithm to generalized continued fractions might give another way of solving it.

The second "catch" with Gosper's algorithm is that the entries of the coefficient cube, which determine the effects the next input partial quotients have on the output, can grow arbitrarily large as more and more partial

quotients are ingested from x and y . This problem is discussed in Section 5.2, particularly Subsection 5.2.4, and in Chapter 6. It also inspired some of the questions for future research in Chapter 7.

Chapter 5

Alternatives in the Literature

This chapter describes the following proposed representation systems for computer arithmetic: The fixed-slash and floating-slash representations by Matula and Kornerup [KM81,KM83a,MK85]; the binary-coded lexicographic continued fraction (LCF) representation by Matula and Kornerup [MK83, KM85,KM87,KM88]; the hybrid fixed-slash and floating-point representation by Hwang and Chang [HC78]; the variable-length-exponent representation by Iri and Matsui [MI81]; the repeating-mantissa floating-point representation by Yoshida [Yos83]; the hyper-exponential representation by Olver and Clenshaw [Olv87]; and the finite p -adic representation by Gregory and Krishnamurty [GK84]. Our descriptions of each alternative include comments on its fitness as a scheme for representing the endpoints of intervals, and whether it facilitates parallel computation.

We argue that the variable-length-exponent representation would be better for command and control applications than standard floating-point representations. This and the other representations form the basis for our new representation proposals in Chapter 7.

5.1 Fixed-Slash and Floating-Slash

This section defines the fixed-slash and floating-slash representations systems proposed by Matula and Kornerup [KM81,KM83a,MK85]. Both sys-

tems represent rational numbers as numerator-denominator pairs stored in fields containing a fixed number of bits. A hypothetical "slash", similar to the slash that ordinarily indicates division and separates numerators from denominators in fractions such as $56501575/103247889$, separates the numerator in each field from the denominator. In the fixed-slash representation the slash is always fixed at the center of the field of bits. In the floating-slash representation the slash's position is determined by a separate slash-position integer, and can be at any position in the field of bits and also at "positions" outside the ends of the field.

5.1.1 Representations of Numbers

More specifically, in a $(2k + 2)$ -bit fixed-slash system, a value consists of a sign bit, k numerator bits, an exact/inexact bit, and k denominator bits. The numerator and denominator fields give nonnegative integers p and q , respectively, in binary. If $s \in \{0, 1\}$ is the sign bit, the value represents the rational $(-1)^s p/q$. The value is in *normal form* if $\gcd(p, q) = 1$.

The rationals representable in a $(2k + 2)$ -bit fixed-slash system are exactly the $n = 2^k - 1$ order- n Farey fractions, where F_n is defined by

$$F_n = \left\{ \pm \frac{p}{q} : 0 \leq p, q \leq n, \gcd(p, q) = 1 \right\}.$$

It is easy to see that, since $+1/0 = +\infty$ is a representable value, the nonnegative members of F_n form a simple chain for any $n \geq 1$, and every member of F_n is either a member or the negative of a member of this simple chain.

Though we will not give them here, Matula and Kornerup define special combinations with 0 in the numerator, denominator or both to represent ± 0 , $\pm \infty$ and not-a-number *NaN* values similar to those in IEEE floating-point arithmetic [IEE85]. The exact/inexact bit is used to record whether the rational stored is the exact result of the operation that produced it or an approximation to this result.

In a $(k + m + 1)$ -bit floating-slash system, with $m \geq \log_2 k$, a value consists of a sign bit, an exact/inexact bit, a $(k - 1)$ -bit *fraction field* giving a concatenated numerator and denominator-with-leading-bit-deleted pair, and an m -bit signed binary integer giving the slash position. Let $s \in \{0, 1\}$ be

the sign bit, let f be the value in the fraction field, and let e be the integer stored in the m -bit slash position. If e is not the largest integer representable in m bits, the number y represented by a $(k + m + 1)$ -bit floating-slash value is given by $y = (-1)^s \cdot p/q$, where the positive integers p and q are determined from e and f as follows: If $0 \leq e \leq k - 2$, p is the binary integer given by the first $k - 1 - e$ bits of f , and q is the binary integer given by 1 concatenated with the final e bits of f . If $e < 0$, p is the binary integer given by 1 concatenated with all $k - 1$ bits of f and with a string of $|e|$ 0's, and q is 1. If $e \geq k - 1$, p is 1 and q is the binary integer given by 1 concatenated with all $k - 1$ bits of f and with $e - (k - 1)$ 0's. The value is in *normal form* if $\gcd(p, q) = 1$.

If e is the largest integer representable in m bits, the value represented is either $\pm\infty$ or NaN . Though we will not give them here, Matula and Kornerup list specific interpretations for e and f in this case to distinguish the separate possibilities. As in fixed-slash, the exact/inexact bit in floating-slash distinguishes exact rationals from approximating ones.

To give simple examples, let $k = m = 5$ and ignore the sign and exact/inexact bits. The field f then contains $5 - 1 = 4$ bits. If the slash position is given by a signed binary integer, the possible values of e range from -15 to 15. If $e = 15$, the value coded is $\pm\infty$ or NaN . Several examples for $e < 15$ follow:

$e = 1$	$f = 1001$	$p = 100_2 = 4$	$q = 11_2 = 3$
$e = 2$	$f = 0110$	$p = 01_2 = 1$	$q = 110_2 = 6$
$e = 0$	$f = 1001$	$p = 1001_2 = 9$	$q = 1$
$e = -2$	$f = 1010$	$p = 1101000_2 = 104$	$q = 1$
$e = 4$	$f = 1001$	$p = 1$	$q = 11001_2 = 25$
$e = 6$	$f = 1010$	$p = 1$	$q = 1101000_2 = 104$

Actually, we have described only one of the possible encodings for floating-slash values suggested by Matula and Kornerup. They note in [MK85] that the operations on floating-slash values might be speeded up by putting the denominator bits before the numerator bits and putting them in reverse order.

Let the *extended* $(k + m + 1)$ -bit floating-slash numbers be the rationals representable by $(k + m + 1)$ -bit floating-slash values with slash-position values e such that $e < 0$ or $e > k - 2$. If $e < 0$, an extended $(k + m + 1)$ -bit floating-slash number is identical to a base-2 floating-point number with

an implicit leading 1 and $k - 1$ more bits in its mantissa. If $e > k - 2$, an extended $(k + m + 1)$ -bit floating-slash number is very similar to such a base-2 floating-point number with a negative exponent. The floating-slash representation thus has floating-point's capacity to conveniently represent numbers of widely varying magnitudes.

Let FSL_k , the *standard* floating-slash numbers with $(k - 1)$ -bit fraction fields, be the union of $\{\pm 0/1, \pm 1/0\}$ and the set of rationals representable by $(k + m + 1)$ -bit floating-slash values with slash-positions e in the range $0 \leq e \leq k - 2$. It is easy to check that the nonnegative members of FSL_k form a simple chain.

It is also easy to check that FSL_k is closely related to the $n = 2^{k-1} - 1$ set of *order- n hyperbolic fractions* H_n defined by

$$H_n = \left\{ \pm \frac{p}{q} : pq \leq n, \gcd(p, q) = 1 \right\}.$$

Specifically,

$$FSL_{k-1} \subset H_{2^{k-1}-1} \subset FSL_k.$$

The nonnegative members of H_n also form a simple chain, and FSL_k can be approximated by $H_{2^{k-1}-1}$ with the loss of less than one bit of representation capacity. Thus while the floating-slash values are dependent on the choice of 2 as a base, a set of standard floating-slash numbers is very similar to a set of hyperbolic fractions, so is almost base-independent.

The loss of storage efficiency for both the fixed-slash and floating-slash representations created by the presence of nonnormalized values is minimal. Let F_n^+ be the positive order- n Farey fractions, and for a set X let $|X|$ denote the cardinality of X . By results of Dirichlet [MK85],

$$\lim_{n \rightarrow \infty} |F_n^+|/n^2 = 6/\pi^2 \approx 0.6079$$

and

$$\lim_{n \rightarrow \infty} \frac{|H_n|}{|\{\pm p/q : pq \leq n, p \geq 1, q \geq 1\}|} = 6/\pi^2 \approx 0.6079.$$

The loss from nonnormalized values is thus less than one bit.

5.1.2 Arithmetic Operations

For fixed-slash and standard floating-slash numbers, arithmetic is performed as described earlier: The result of any operation is computed as if the operation were first performed exactly and then mediant rounding was used to round the result to a representable value. Matula and Kornerup give algorithms for this arithmetic in [KM83a]. It is actually not necessary to compute the exact result of an operation first and then apply a mediant rounding algorithm; one can get the effect of this by initializing p_{-2} , q_{-2} , p_{-1} and q_{-1} in the mediant rounding algorithm (see Section 4.3) to values determined by the binary operation and one of its arguments, and then applying the algorithm to compute "convergents" of the other argument. See [KM83a] for details. This means of doing arithmetic is essentially a special case of the algorithm by Gosper described in Section 4.5.

Matula and Kornerup [KM83a] give algorithms using both the standard continued fraction and the generalized continued fraction versions of mediant rounding described in Sections 4.3 and 4.4. Both of these algorithms can be made to perform arithmetical operations at the same time they deduce proper roundings by making appropriate initializations. Matula and Kornerup show how these algorithms can be implemented with binary shift and add/subtract operations, and note which parts of these algorithms can be performed in parallel. These observations are special cases of, and inspired, their ideas on how arithmetic could be performed in LCF that will be described in Subsection 5.2.3.

These algorithms produce correct, but possibly nonnormalized, results if their arguments are not normalized. It is thus not necessary to normalize arguments before performing arithmetic operations.

Matula and Kornerup also suggest how the add/subtract operations might be speeded up by using carry-save or borrow-save arithmetic, techniques that use redundancy in the representation of integers. The algorithms for performing arithmetic and doing rounding require being able to normalize integers and determine whether they are zero, operations that are generally difficult for redundantly-coded integers. Matula and Kornerup give techniques for reducing the zero-determination problem to the normalization one, and for doing the necessary integer normalizations for both carry-save and borrow-save representations by looking only at the first two digits [KM83a]. If

redundantly-coded integers are used, integer results must be converted back into standard binary to produce the final fixed- or floating-slash results.

Although arithmetic units for fixed-slash and standard floating-slash representations have not yet been implemented in hardware, Matula and Kornerup give a theoretical analysis of the numbers of operations and sub-operations that must be performed in carrying out their algorithms that suggest the times needed to do fixed-slash and standard floating-slash arithmetic are comparable to the times needed to do noniterative divide operations on binary integers with similar numbers of bits [KM83a]. They also found results consistent with these estimates in simulation experiments.

Matula and Kornerup do not give explicit algorithms for doing arithmetic on pairs of extended floating-slash values or pairs containing a standard and an extended floating-slash value. Presumably, computed results could always be produced as if the operations were first performed exactly and nonzero exact results were rounded as follows:

- If the magnitude of the exact result is between that of the smallest and largest positive standard floating-slash numbers, use median rounding to round to a standard floating-slash number.
- Otherwise, round the exact result to the nearest extended floating-slash number if this number is determined uniquely, and round it to the nearest extended floating-slash number whose final digit is 0 if there are two equally-near such numbers.

We do not know, however, how making such an extension to the arithmetic unit proposed by Matula and Kornerup would affect the unit's speed and complexity.

5.1.3 Errors in Median Rounding

The gap sizes between consecutive representable numbers for both fixed-slash and standard floating-slash representations vary greatly, so the amount of error introduced by median rounding can also vary greatly. We will follow Matula and Kornerup in approximating the fixed-slash and standard floating-slash representations by Farey and hyperbolic fractions, respectively. By the

results on simple chains given earlier, if p/q and p'/q' are two consecutive Farey or hyperbolic fractions then

$$p'/q' - p/q = 1/qq', \quad \text{so} \quad \frac{p'/q' - p/q}{p/q} = 1/pq'.$$

For the order- n Farey fractions, the gap sizes for consecutive fractions in the interval $[0, 1]$ thus vary from about $1/n^2$ to about $1/n$, and for the order- n hyperbolic fractions the relative gap sizes for consecutive finite fractions vary from about $1/n$ to about $1/\sqrt{n}$.

The absolute error introduced by mediant rounding on the interval $[0, 1]$ in a $(2k+2)$ -bit fixed-slash system can thus be as large as about 2^{-k} , and the relative error introduced by mediant rounding in a $(k+m+1)$ -bit floating-slash system can be as large as about $2^{-k/2}$. These error values are far worse than the corresponding absolute error of about 2^{-2k} for $2k$ -bit binary fixed-point numbers and the corresponding relative error of about 2^{-k} for k -bit binary floating-point numbers. However, as we will now show, typical absolute and relative error values introduced by mediant rounding are much smaller and are comparable to the errors in ordinary fixed-point and floating-point arithmetic. Thus, even though mediant rounding tends to produce errors that are larger than those for round-to-nearest rounding in computations whose ideal results are not rational, the loss can be expected to be small.

Typical errors introduced by mediant rounding are much smaller than maximum ones because typical gap sizes between consecutive Farey or hyperbolic fractions are much closer to the lower extremes than the upper ones. Large gap sizes only occur around comparatively rare simple fractions. Further, around these simple fractions the large gap sizes give mediant rounding its desirable bias towards simplicity.

If R is a set of rationals consisting of the members of a simple chain and their negatives, and Φ_R is the mediant rounding function rounding reals to members of R , Matula and Kornerup show the following [MK80]:

- If $R = F_n$ and X is a uniformly-distributed random variable on $[0, 1]$,

$$\text{Exp}(|X - \Phi_R(X)|) = \frac{6}{\pi^2} \frac{\log n}{n^2} + O\left(\frac{1}{n^2}\right)$$

and for $1 \leq \alpha \leq 2$,

$$\text{Prob}\{|X - \Phi_R(X)| > n^{-\alpha}\} \leq 2n^{\alpha-2}.$$

- If $R = H_n$ and X is a log-uniformly-distributed random variable on $[1/n, n]$,

$$\text{Exp}\left(\frac{|X - \Phi_R(X)|}{X}\right) < \frac{\log n}{2n}$$

and for $1/2 \leq \alpha \leq 1$ and n sufficiently large,

$$\text{Prob}\left\{\frac{|X - \Phi_R(X)|}{X} > n^{-\alpha}\right\} < 4\alpha n^{\alpha-1}.$$

Absolute and relative error values are thus closely distributed around expected values close to the error values for corresponding binary fixed-point or floating-point numbers with the same number of bits of precision. Empirical results on average and "better than all except one in a million or one in a trillion" gap sizes and relative gap sizes, for implementations of fixed-slash and floating-slash representations, are consistent with these expectations [MK85].

5.1.4 Empirical Results

Matula and Ferguson [FM85] give empirical information strongly related to mediant rounding and standard floating-slash arithmetic. They produce this information with a floating-point simulator that computes an approximation to Φ_R for R a set of hyperbolic fractions. They use this simulator to perform two sets of complicated calculations, one for ideal numbers which are all rationals and the other for ideal numbers which are all irrationals. They compare the results for these calculations, using the simulator to perform the arithmetic operations, with the results for these calculations produced using ordinary floating-point arithmetic.

Although they are intended to test floating-slash arithmetic, the results actually compare two different ways of using floating-point hardware. The results are stated not with respect to ideal numbers, but with respect to the best-possible approximations to these ideal numbers for the version of

floating-point arithmetic being used. They show, among other things, that doing a floating-point simulation of mediant rounding can produce significantly more accurate answers for calculations involving quantities whose ideal values are simple rationals.

The simulator represents a fraction p/q as the floating-point result of dividing the integer p by the integer q . It simulates approximate rational arithmetic by using a floating-point approximation to the rational arithmetic algorithm described in Section 4.3. To perform addition, say, it starts with two of its floating-point representations for rationals, computes the floating-point sum v of these representations, uses floating-point arithmetic to compute (approximations to) the partial quotients and convergents of v , finds the last convergent p_i/q_i such that $p_i/q_i < m$ for an integer limit m , and returns the floating-point result of dividing p_i by q_i as the result of the addition.

Note that since partial quotients are integers, many of the partial quotients and convergents the simulator computes are exactly correct. Note further, though, that the intermediate result v is only an approximation to the ideal result of performing the desired operation on the pair of rationals represented by the floating-point values combined to produce v , and the errors in floating-point inversion will also eventually cause the computed partial quotients for v to be incorrect.

The simulator produced the empirical results in [FM85] by running on a CDC 6600. It used single-precision arithmetic in some tests, and double-precision arithmetic in others. Since the floating-point values on the CDC 6600 have 48-bit mantissas in single-precision and 96-bit mantissas in double-precision, the simulation took $m = 2^{48}$ for the single-precision calculations and $m = 2^{96}$ for the double-precision ones. In all cases it thus computed convergents until the product of their numerator and denominator was no longer exactly representable in the floating-point arithmetic currently being used.

The computations involving only numbers whose ideal values are rational were finding the inverses of Hilbert matrices. The order- n Hilbert matrix H_n is the n by n matrix whose entry in position i, j , $1 \leq i, j \leq n$, is $1/(i+j+1)$.

For example,

$$\mathbf{H}_3 = \begin{bmatrix} 1/1 & 1/2 & 1/3 \\ 1/2 & 1/3 & 1/4 \\ 1/3 & 1/4 & 1/5 \end{bmatrix}.$$

Hilbert matrices arise in finding the best least-squares approximation, over the interval $[0,1]$, of a continuous function by a polynomial of a given degree [FM85].

The problem of inverting Hilbert matrices is often used as a test of arithmetic systems (c.f., the comments on matrix inversion in [KL85]) because it can be solved exactly, using formulas included in [FM85], and is very ill-conditioned. The entries of \mathbf{H}_n^{-1} are integers for all n , and the entry with the largest magnitude increases very rapidly as n increases — the largest entry in \mathbf{H}_5^{-1} , for example, is 179200. For any matrix A , let $\max(A)$ be the magnitude of the element of A for which this magnitude is maximum. As Matula and Ferguson explain in [FM85], if A is an n by n matrix, the *condition number* $n \cdot \max(A) \cdot \max(A^{-1})$ estimates how greatly an initial relative error in an entry in A is magnified into a final relative error in an entry in a computed solution to $AX = B$.

The base-10 logarithm of A 's condition number thus estimates how many base-10 digits are likely to be lost in the computed final entries of A^{-1} ; lost in addition to those digits already lost by approximating these entries and the entries of A by floating-point numbers of a particular precision. (These error estimates could probably have been stated more clearly in terms of mantissa bits lost in the computed results.) Using estimates in the literature, Matula and Ferguson estimate that about $1.53n$ decimal digits of accuracy can be expected to be lost in computing the inverse of \mathbf{H}_n . Since the largest entry in \mathbf{H}_n is 1, this estimate is equivalent to estimating the greatest magnitude of an entry in \mathbf{H}_n^{-1} as $10^{1.53n}$.

The algorithm Matula and Ferguson use for computing the inverse of an n by n matrix A , a method recommended as reducing computational error, treats the identity $AA^{-1} = I$ as a collection of n systems of linear equations. It solves each system as follows: It uses Gaussian elimination with no pivoting to compute a factorization $A = LU$ of A , where L and U are lower- and upper-triangular matrices respectively. It then solves $LY = B$ by forward elimination, and solves $UX = Y$ by backward substitution.

In their results, Matula and Ferguson give estimated and observed numbers of decimal digits of accuracy lost in computing the inverses of Hilbert matrices of different orders. The estimated digit-loss numbers are those given by the condition numbers, and the observed digit-loss numbers are calculated from the relative errors, scaled to fit the precision of the floating-point arithmetic being used, in the entries of the computed matrices. They give results for calculations using four different types of arithmetic:

1. Single-precision floating-point;
2. Double-precision floating-point;
3. Simulated approximate rational arithmetic, single-precision floating-point and $m = 2^{48}$; and
4. Simulated approximate rational arithmetic, double-precision floating-point and $m = 2^{96}$.

The results, given in Figure 4 in [FM85], are striking. The observed numbers of digits lost using straightforward floating-point arithmetic, both for single- and double-precision, closely match the losses predicted by the condition numbers — in both cases the observed loss numbers vary from about 1 to 4 fewer digits lost, but these differences are for estimated numbers of digits lost as large as 29. The numbers of digits lost using simulated approximate rational arithmetic are much lower. The single-precision simulation inverts each of H_1 through H_9 with the loss of less than a single digit. The double-precision simulation inverts each of H_1 through H_{19} with the loss of less than a single digit; for H_{19} , by contrast, straightforward double-precision arithmetic loses about 25 of the roughly 29 digits available in double-precision floating-point.

Note that by the estimate used for the condition numbers, the magnitude of the largest element in H_{19}^{-1} is on the order of $2^{96.6}$ and the magnitude of the largest element in H_9^{-1} is on the order of $2^{45.7}$. Both simulations thus only begin to lose accuracy in inverting the Hilbert matrices when the inverses begin to contain elements that are not exactly representable in the floating-point arithmetic being used.

The simulations' observed numbers of digits lost increase sharply beyond these limits, going to about 7 for single-precision on H_{10} and to about 10 for

double-precision on H_{20} , but the single- and double-precision simulations succeed in inverting all the Hilbert matrices through H_{15} and H_{29} , respectively, before their numbers of digits lost exhaust all the digits available for their respective forms of floating-point arithmetic. The straightforward single- and double-precision floating-point calculations only retain some of the available digits for matrices no larger than H_{13} and H_{20} , respectively.

Matula and Ferguson also give estimated and observed numbers of decimal digits lost in computing the inverses of the matrices $H'_n = DH_nD$ for D a diagonal matrix whose i th entry is the i th root of a randomly-chosen, 108-binary-digit number in the interval $(0,1)$. They give results for the worst cases with 25 different choices of the initial random number. The condition number of an H'_n is the same as that of H_n , but its ideal entries are irrational. The estimated number of digits lost for inverting an H'_n is the same as the number for inverting H_n . Matula and Ferguson give observed digit-loss results for calculations using straightforward double-precision arithmetic and simulated approximate rational arithmetic, computed with double-precision floating-point and $m = 2^{96}$. In the simulated rational arithmetic calculations, the initial entries of the H'_n are rounded to rationals p/q such that $pq \leq 2^{96}$.

The results, given in Figure 5 in [FM85], show that the actual numbers of digits lost for both floating-point and simulated approximate rational arithmetic calculations are virtually identical, and both are virtually identical to the estimated numbers of digits lost given by the condition numbers. The floating-point results are typically better by a fraction of a decimal digit. These results, together with the results on inverting Hilbert matrices, strongly support the assertion that for values stored in comparable numbers of bits, floating-slash arithmetic would produce much better results than floating-point on calculations whose ideal values are rational, and would produce comparable results on calculations whose ideal values are irrational.

5.1.5 Potential Applications

The theoretical and empirical results given above both indicate that floating-slash arithmetic, particularly floating-slash arithmetic that includes extended floating-slash values, would perform as well as floating-point for typical applications and perform significantly better than floating-point for applications

doing calculations on rational numbers. The chief disadvantage of floating-slash is that it is slower, though its other disadvantages — not being appropriate for interval arithmetic, and presumably producing computational errors that are harder to analyze and bound — could also be significant. As a practical matter, hardware doing floating-point arithmetic is also widely available, while hardware doing floating-slash arithmetic is just now being developed.

Matula and Kornerup [MK85] list these potential applications for systems doing approximate rational arithmetic:

- Symbolic computation programs that mix exact rational and approximate real arithmetic;
- Combinatorial optimization problems, as in linear programming with sparse 0-1 constraint matrices; and
- Knowledge-based systems applications in which it is critical to recognize simple rationals.

We were unable to find applications in command and control software where rational numbers arose to a significant extent. In particular, they did not arise in the fragment of the Hostile Booster Interception code [App87] that the Reals project examined, and we learned of no such applications through our inquiries with our Air Force contract monitors and with experts in Operations Research.

Floating-slash arithmetic thus seems to be a number representation system with advantages over floating-point, but advantages that have not yet been obvious enough to inspire its widespread use. Significantly, Knuth [Knu81, page 316] lists the following as an exercise of “significant open problem level” difficulty:

Modify one of the compilers at your installation so that it will replace all floating-point calculations by floating-slash calculations. Experiment with the use of slash arithmetic by running existing programs that were written by programmers who actually had floating-point arithmetic in mind. . . . Are the results better or worse when floating-slash numbers are substituted?

Note further that the simple simulator program used by Matula and Ferguson [FM85], and described above, captures much of the advantage of mediant rounding, with mediant rounding's bias in favor of simple results, for computations on rational numbers, and does it with existing floating-point hardware. Although a hardware implementation of floating-slash arithmetic might produce still more accurate results — the simulator computes only an approximation to floating-slash arithmetic — the main advantage of such hardware would be its greater speed. For computations involving rational numbers for which speed is not a factor, the simulator will presumably suffice.

Neither the fixed-slash nor floating-slash representation is appropriate as a means for representing the endpoints of intervals, and neither facilitates parallel computation, though parts of individual operations can be done in parallel.

5.2 The LCF Representation System

This section defines the Lexicographically-Coded Continued Fraction (LCF) representation by Matula and Kornerup [MK83,KM85,KM87,KM88]. This representation gives a binary encoding of finite initial segments of real numbers' standard continued fractions; for the remainder of this section, we will assume all continued fractions are standard unless we specifically note otherwise. If the number of bits available for each binary encoding is fixed, the LCF representation becomes a form of approximate rational arithmetic. The encoding has the convenient property that the lexicographic order on real numbers' binary encodings is equivalent to numerical order on the real numbers, hence the representation's name.

With the LCF representation, arithmetic can be performed in an on-line fashion. (On-line arithmetic was described in Section 3.4.) The LCF representation thus facilitates parallel computation, and combines the advantages of on-line arithmetic with those of approximate rational arithmetic. Further, the LCF representation supports performing computations to a specified precision. Representation systems giving on-demand accuracy, and carrying out calculations to precisions determined by the numbers being calculated, will be considered in Chapter 6.

This section defines the LCF representation, discusses its efficiency as a means of storing real numbers, and describes how arithmetic can be performed on numbers given in their LCF forms. It ends with a brief discussion of how redundantly-coded binary numbers or generalized continued fractions might be used to avoid a throughput problem that can arise in LCF arithmetic.

5.2.1 The LCF Encoding

We will present the LCF encoding by first giving an encoding for positive integers, then an encoding for the continued fractions of nonnegative rationals, then an encoding for arbitrary continued fractions, and finally an encoding for arbitrary real numbers. All encodings are in binary, and are given by Matula and Kornerup in [MK83].

Every positive integer m has a base-2 representation

$$m = (1b_{n-1}b_{n-2}\cdots b_0)_2 = 2^n + b_{n-1}2^{n-1} + \cdots + b_0.$$

The bit string $1^n 0 b_{n-1} b_{n-2} \cdots b_0$, where 1^n denotes n consecutive 1's, then gives a self-delimiting binary encoding of m ; the code for 1 can be taken to be 0, and $+\infty$ can be taken to be an infinite string of 1's. Since these codes are self-delimiting, it is possible to concatenate the codes for a sequence of positive integers into a single bit-string, then unambiguously deduce the sequence of integers from this string. Note that the lexicographic order on these codes matches the numerical order on the corresponding integers.

Every nonnegative rational has a continued fraction $[q_0, \dots, q_k]$ such that $q_0 \geq 0$ and $q_i > 0$ for all $i \neq 0$. This continued fraction can be thought of as $[q_0, \dots, q_k, +\infty]$, so $+\infty$ can be used as an end-marker. As we noted in Section 4.3, the continued fraction can be chosen so that k is even. If i is even, let s_i be the self-delimiting binary code for q_i , and otherwise let it be the 1's complement of the self-delimiting binary code for q_i .

Code the nonnegative rational with continued fraction $[q_0, \dots, q_k]$ as follows: If $q_0 = 0$ — i.e., if the rational is less than 1 — take the first bit of the code to be 0 and take the rest of the code to be the concatenation, for all $1 \leq i \leq k$, of the s_i . If $q_0 > 0$, take the first bit of the code to be 1 and take the rest of the code to be the concatenation, for all $0 \leq i \leq k$, of the s_i .

Note that the code for 0 is 0, and that the code for every rational ends with an infinite string of 0's. Further note that since increasing an even-indexed partial quotient of a (standard) continued fraction makes the corresponding real larger, while increasing an odd-indexed partial quotient makes that real smaller, the lexicographic order on these codes matches the numerical order on the corresponding rationals.

For an arbitrary rational r , if $r \geq 0$ code r as 1 followed by the coding just given of $|r|$, and if $r < 0$ code r as 0 followed by the 2's complement of the coding just given of $|r|$. (Taking 2's complements is necessary to have the codes always end with infinite strings of 0's.) Lexicographic order on the codes still matches numerical order on the corresponding rationals.

Here are several examples, taken from [MK83], of this coding for arbitrary rationals. They illustrate why various aspects of the coding are necessary. Every code ends with an infinite string of 0's.

8	11111	-1/8	01111
4	11110	-1/4	01110
3	11101	-1/3	01101
2	11100	-1/2	01100
5/3	11011	-3/5	01011
3/2	11010	-2/3	01010
1	11000	-1	01000
2/3	10110	-3/2	00110
3/5	10101	-5/3	00101
1/2	10100	-2	00100
1/3	10011	-3	00011
1/4	10010	-4	00010
1/8	10001	-8	00001
0	10000	$\pm\infty$	00000

Finally, for an arbitrary real, code the real as the limit of the codes for the initial segments, rewritten if necessary to make their final indexes even, of that real's continued fraction. Lexicographic order on the codes matches numerical order on the corresponding reals.

5.2.2 LCF Gap Sizes and Range

We will present theoretical and empirical results from [KM85] on the sizes of the gaps between consecutive values representable in an LCF system for which there is a fixed bound on the number of bits in an LCF code. By abuse of terminology, we will use "the LCF encoding" to refer, depending on the context, to either the encoding of positive integers, the encoding of nonnegative rationals, or the encoding of arbitrary rationals just described. Note that the bit-length of a positive integer's LCF code increases by 1 if the integer is interpreted as a nonnegative rational, and the bit-length of a nonnegative rational's LCF code increases by 1 if the nonnegative rational is interpreted as an arbitrary rational.

If a rational p/q is chosen randomly and uniformly over $[0, 1]$, for sufficiently large i the probability that the i th partial quotient of this rational's partial fraction is j is [Knu81]

$$\log_2 \frac{(j+1)^2}{j(j+2)}.$$

By information-theoretic arguments, an optimal coding for this frequency distribution would use

$$-\log_2(\log_2 \frac{(j+1)^2}{j(j+2)})$$

bits to code j . With the LCF encoding, the expected number of bits per partial quotient for rationals chosen uniformly over $[0, 1]$ is then 3.52, while for the optimal coding the expected number of bits per partial quotient would be 3.45 [KM85].

In particular, the LCF code for 1 is 0, which requires only 1 bit, while an optimal code would use 1.27 bits. The LCF codes for 2 and 4 are 100 and 11000 respectively, requiring 3 and 5 bits, while an optimal code would use only 2.56 and 4.09 bits for these numbers. For LCF codes with only k bits, in the interval $[0, 1]$ the smallest gap sizes between consecutive LCF values can thus be expected near the rational with continued fraction $[0, 1, 1, 1, \dots, 1]$ and LCF code 01010101...01, while the largest gap sizes can be expected near the rational with continued fraction $[0, 2, 4, \dots, 2, 4]$ and LCF code 00111100001111 ... 00001111; in the first case the gap sizes grow asymp-

totically as $2^{-1.3885k}$, and in the second case they grow asymptotically as $2^{-0.8268k}$ [KM85].

Matula and Kornerup give exhaustive or Monte Carlo analyses of gap sizes on $[0, 1]$ for k -bit LCF codes, and the maximum and minimum gap sizes are consistent with these expectations. In all cases, gap sizes vary between $2^{-1.38k}$ and $2^{-0.82k}$, and keep this range of variation even as k increases. However, as k increases the gap sizes become log-normally distributed around $2^{-1.0k}$, exactly the gap size for 2^k uniformly-distributed points in $[0, 1]$. For $k = 128$, 99.9% of the gap sizes are between $2^{-1.12k}$ and $2^{-0.90k}$. Thus even though the extreme gap sizes persist as k increases in being exponentially larger or smaller than the gap size for uniformly-distributed points, typical gap sizes become closer to those for uniformly-distributed points [KM85]. The k -bit LCF codes are *not* asymptotically uniformly dense on $[0, 1]$ as the k -bit fixed-slash values are; this shows the dependence of LCF codes on the base 2.

LCF codes also do not have the ability to represent numbers of greatly varying magnitudes in fixed numbers of bits, one of the great advantages of the floating-point and extended floating-slash representations. Matula and Kornerup suggest using a separate, independent representation of the large-magnitude partial quotient q in continued fractions of the forms $[q, \dots]$, $[0, q, \dots]$ and $[-1, 1, q, \dots]$ to extend the range of representable values [MK83]. Another possibility, similar to one of the means for representing rationals given in our interim report (see Section 4.1), is to represent reals as fractions in the interval $(-1, 1)$ multiplied by powers of 2; the fractions could be given by LCF codes and the powers by an integer exponent. We will note how arithmetic operations might be performed on numbers represented in this way in the next subsection.

5.2.3 LCF Arithmetic

As we described earlier, in Gosper's algorithm the necessary updates to the coefficient and decision cubes can be made by multiplying by matrices of the forms

$$\begin{bmatrix} 0 & 1 \\ 1 & p \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ p & p-1 \end{bmatrix}, \text{ and } \begin{bmatrix} 0 & 1 \\ 1 & -p \end{bmatrix}$$

for nonnegative integers p . If $p > 0$, by the base-2 representation of p , $p = 2^n + b_{n-1}2^{n-1} + \dots + b_0$ for some collection of b_i , with each $b_i \in \{0, 1\}$.

These matrices can be factored as follows:

$$\begin{bmatrix} 0 & 1 \\ 1 & p \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}^n \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1/2 & b_{n-1}/2 \\ 0 & 1 \end{bmatrix} \dots \begin{bmatrix} 1/2 & b_0/2 \\ 0 & 1 \end{bmatrix},$$

$$\begin{bmatrix} 1 & 1 \\ p & p-1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix}^n \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \frac{1+b_{n-1}}{2} & \frac{b_{n-1}}{2} \\ \frac{1-b_{n-1}}{2} & \frac{2-b_{n-1}}{2} \end{bmatrix} \dots \begin{bmatrix} \frac{1+b_0}{2} & \frac{b_0}{2} \\ \frac{1-b_0}{2} & \frac{2-b_0}{2} \end{bmatrix},$$

and

$$\begin{bmatrix} 0 & 1 \\ 1 & -p \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}^n \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1/2 & -b_{n-1}/2 \\ 0 & 1 \end{bmatrix} \dots \begin{bmatrix} 1/2 & -b_0/2 \\ 0 & 1 \end{bmatrix}.$$

Note that there is a one-to-one correspondence between the factors in these factorizations and the bits in the LCF code for p . In Gosper's algorithm, it is thus possible to ingest individual *bits* of the LCF codes for the partial quotients of x and y , and output individual *bits* of the LCF codes for the partial quotients of z . It is only necessary to maintain status flags recording things such as whether the partial quotient being ingested has an even or odd index to keep track of whether each bit ingested should be interpreted as itself or its negation and whether each bit output should be output as itself or its negation. It is thus possible to do approximate rational arithmetic at the bit level, and produce more accurate results if they are demanded just by outputting more bits.

Every input bit corresponds to a shift or shift-and-add-or-subtract operation. If $p = 0$, the necessary updates can also be made with a single such operation. Further, four of the eight necessary coefficient updates can be performed in parallel [KM88].

In producing output, it is not necessary to ingest inputs until the next partial quotient of z is determined, but only until the next bit of the LCF encoding of the next partial quotient of z is determined. This generally makes it possible to produce output much earlier. (This method of improving throughput is more natural than the idea suggested by Jones [Jon84] mentioned in our interim report; Jones suggests giving partial quotients as sequences of nested intervals and using these intervals to get tighter bounds on the extreme values of z .)

A general description of the extension of Gosper's algorithm to LCF codes follows. A more detailed description is in [KM88]. For simplicity, our description assumes that both x and y are positive and says "output 0", say, instead of "output whichever of 0 or 1 would be interpreted as the bit 0 in the LCF code for the partial quotient of z currently being produced".

1. Initialize the counter c to 0.
2. Input bits from x and y until it is possible to determine that $z \geq 2$ or that the first partial quotient of z is 1 — i.e., that

$$2 \leq A/E, B/F, C/G, D/H$$

or that

$$1 \leq A/E, B/F, C/G, D/H < 2.$$

In the first case, output 1, perform the

$$\begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$$

transformation for output to z , increment the counter c , and repeat step 2. (This has the effect of dividing z by 2 and continuing.) In the second case, output 0, perform the

$$\begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}$$

transformation for output to z , and go to step 3.

3. While $c > 0$, decrement c , then input bits from x and y until it is possible to determine that $z \geq 2$ or that the first partial quotient of z is 1. In the first case, output 0 and perform the

$$\begin{bmatrix} 1/2 & 0 \\ 0 & 1 \end{bmatrix}$$

transformation for output to z ; in the second case, output 1 and perform the

$$\begin{bmatrix} 1/2 & -1/2 \\ 0 & 1 \end{bmatrix}$$

transformation for output to z . After $c = 0$, go back to step 2 to produce the next partial quotient of z .

Matula and Kornerup give estimates of the average numbers of partial quotients in the continued fractions for rationals in the set H_n of hyperbolic fractions. They also give estimates of the numbers of shift and add/subtract operations that must be performed per partial quotient for arithmetic operations that take members of H_n as inputs and produce members of H_n as outputs [KM87]. These estimates are based on the assumption that bits are output at roughly the same rate they are ingested, an assumption that is not always correct and is discussed more fully in the next subsection.

Since Gosper's algorithm can compute operations other than addition, subtraction, multiplication and division by making appropriate initializations of the coefficient cube, it could be used to perform arithmetic operations on reals given as rationals in the interval $(-1, 1)$ times a power of 2. For example, for reals given as $x \cdot 2^i$ and $y \cdot 2^j$, where x and y are given by LCF codes, Gosper's algorithm could be used to compute $x + y$ by initializing the coefficient cube to $a = 0$, $b = 2^i$, $c = 2^j$, $d = 0$, $e = 0$, $f = 0$, $g = 0$ and $h = 1$. Questions related to this approach are noted in Chapter 7.

5.2.4 Possible Uses of Redundancy

Although the extension of Gosper's algorithm to LCF codes is usually able to output the next bit of z after ingesting only a few bits of x and y — in simulations, Matula and Kornerup found typical delays of about 5 bits [KM87] — there are situations where potentially infinite numbers of bits must be ingested from x and y to determine the next output bit for z . Matula and Kornerup [KM88] give an example where the result is either slightly greater than 2, with LCF form $11000 \dots 01 \dots$, or slightly less than 2, with LCF form $10111 \dots 10 \dots$. This example is equivalent to not being able to decide whether a number's continued fraction is $[2, k]$ or $[1, 1, m]$ for large integers k and m .

This is exactly the sort of problem that arises with nonredundant representations of numbers. Matula and Kornerup are currently investigating the possibility [KM88] of using the redundant "bit" set $\{0, 1, \bar{1}\}$, where $\bar{1} = -1$, to avoid this problem. In the example, the arithmetic unit could output 1 and "correct" it if necessary later by outputting $\bar{1}$. They hope in this way to guarantee uniform throughput.

As we noted in Section 4.5, the redundancy in generalized continued fractions allows these continued fractions to avoid the “unlimited wait for the next partial quotient” problem in Gosper’s algorithm. They could also be used to guarantee uniform throughput. Related results and questions are given in Chapters 6 and 7.

The second major problem with Gosper’s algorithm, that of how big the entries of the coefficient cube become as the inputs are ingested, also arises for LCF arithmetic. In their simulations, Matula and Kornerup noted that these elements seem to grow about as quickly as bits are ingested from the inputs, so order of k -bit registers for storing the coefficients should work for performing operations on k -bit LCF codes. They list accurately determining the necessary size of these registers as a problem for future research [KM88].

It is possible that using a redundant bit set will not only increase throughput, but also decrease the necessary sizes of the coefficient-cube registers. Our results on the growth of the coefficient-cube entries for versions of Gosper’s algorithm producing standard and generalized continued fractions are described in Subsection 6.4.2; they are essentially contrary to our expectation that increased throughput would reduce the growth of these entries. The problem of trying to limit the sizes of the necessary registers by using other possible encodings also inspired a question for future research in Chapter 7.

5.3 Hybrid Floating-Point and Fixed-Slash

Hwang and Chang [HC78] propose an extension of binary floating-point that allows them to represent many rational numbers exactly. As in ordinary floating-point, they represent each value as a mantissa times a power of 2, and as in ordinary floating-point a mantissa can be a *radix fraction*, or normalized base-2 number in the interval $[1/2, 1)$. Unlike ordinary floating-point, however, the mantissa can also be the ratio of two integers p and q such that $1/2 \leq p/q \leq 1$. If there are $2k$ bits in the field determining the mantissa, Hwang and Chang represent the fraction p/q as two k -bit integers $q - p$ and q . Since it is always true that $q - p < q/2$, with this representation the first bit of the field determining the mantissa is always 0 if the mantissa is a rational and is always 1 if it is a $2k$ -bit radix fraction.

Let R_{2k} be the $2k$ -bit radix fractions, and let

$$U_{2k} = F_{2k} \cup R_{2k}$$

be the union of the $2k$ -bit radix fractions with the Farey fractions F_{2k} . Hwang and Chang prove that there is at most one member of F_{2k} in the gap between two consecutive members of R_{2k} , they define a rounding mapping each real number x in $[1/2, 1)$ to the nearest value in U_{2k} , and they give an algorithm based on computing the continued fraction expansion of x for computing this rounding.

They perform every operation as if its result were first computed exactly, then rounded to the nearest member of U_{2k} . Although they do not mention the possibility, their arithmetic could presumably be extended to support each of the four rounding modes — to-nearest, upward, downward, toward-zero — available in IEEE floating-point arithmetic [IEE85].

They compare relative errors in rounding real numbers in the interval $[1/2, 1)$ to the nearest radix fraction and to the nearest value representable in their system — i.e., they compare relative errors in rounding these reals to the nearest values in R_{2k} and in U_{2k} . For a real number x in this interval, let $\rho(x)$ be the machine-representable value x is rounded to in whichever of R_{2k} and U_{2k} is currently being considered. Let the *relative representation error* for a real number $x \in [1/2, 1)$ be $|(\rho(x) - x)/x|$.

Using uniformly-distributed rationals in the interval $[1/2, 1)$, chosen so that the difference between successive rationals is smaller than the smallest gap in U_{2k} , Hwang and Chang show that for $8 \leq k \leq 20$ the average relative representation error when these rationals are approximated by members of U_{2k} is between 10.2% and 11.4% lower than the average relative representation error when they are approximated by members of R_{2k} . Their representation system thus gives a 10% improvement in relative accuracy, without using any more bits, over a version of floating-point that does not have implicitly-normalized mantissas; a version of floating-point with implicitly-normalized mantissas would have one more bit of precision, with a roughly 50% improvement in relative representation error. Their system also makes many exact rational calculations possible.

Hwang and Chang presume operations in their system will be slower than those in floating-point arithmetic, but propose an arithmetic processor with

a pipelined design that they hope can alleviate this problem. They give no performance results or estimates.

If upward and downward rounding were available in their arithmetic, their representation would only be slightly worse than floating-point with implicitly-normalized mantissas as a means for representing the endpoints of intervals. It does not facilitate parallel computation.

5.4 Variable-Length Exponents

Iri and Matsui [MI81] propose a sensible extension of floating-point arithmetic. Their basic idea is to use binary floating-point numbers with a variable number of bits in the field determining the exponent. In this way, in numbers with moderate exponents more bits can be used to make the mantissa more precise, and in numbers with extreme exponents bits that would otherwise be used for the mantissa can be used to store the exponent. The resulting values are thus more accurate, using the same number of bits, than ordinary floating-point values for numbers of moderate size, and have such a large range that overflow and underflow are virtually impossible.

Iri and Matsui credit Morris [Mor73] with the idea of using a variable-length exponent to make more bits of precision available for moderately-sized numbers; their innovation was using the same concept to make overflow or underflow virtually impossible.

More specifically, they propose for 64-bit values that the first bit code a sign and the final 6 bits code an exponent-length of 0 through 57. If the exponent-length value is n , the n last bits before the final 6 bits code an exponent, including its sign; if $n = 0$ the exponent is 0. The remaining $64 - 1 - 6 - n$ bits code a binary mantissa; if $n = 57$ the mantissa is taken to be 1. If $n < 57$, the mantissa is implicitly normalized to fall in the interval $[1/2, 1)$, so has an implicit leading bit of 1. Values of the exponent-length field from 58 to 63 code infinite and "Not a Number" values.

Actually, the last paragraph defines only level 0 of the representation Iri and Matsui propose. They also propose level-1 values in which the mantissa is taken to be 1 and the value given by the mantissa and exponent fields is the value of the *exponent*. For example, except for the level indicator a

value that codes $0.5 \cdot 2^{17}$ at level 0 codes $1 \cdot 2^{0.5 \cdot 2^{17}}$ at level 1. Iri and Matsui define higher-level values similarly, so that a level- n value codes the exponent of a level- $(n + 1)$ value. Values of the exponent length between 58 and 63 could be used to code the level indicators. Though they did not say so, it is probably true that x , $x + x$ and $x \cdot x$ are indistinguishable for levels no higher than about 6, so underflow and overflow could not occur, as it cannot in Olver and Clenshaw's system to be described in Section 5.6.

Iri and Matsui did not propose specific encodings of higher-level numbers, and did not carry out simulations of arithmetic for them as they did for level 0. We do not believe having higher-level numbers would give any significant advantage over having level 0 with special codes for $\pm\infty$. The positive numbers representable by level 0 range over the extremes

$$2^{\pm 2^{56}} = 2^{\pm 72057594037927936} \sim 10^{\pm 21691497220794363.8},$$

which should be more than enough; as we noted in our interim report, there are estimated to be only about 10^{80} nucleons in the known universe [FH65].

Demmel [Dem87] raises the objection to Iri and Matsui's representation system that it makes writing programs that produce results with a guaranteed maximum relative error when overflow and underflow do not occur more difficult. Demmel also notes, however, that if the hardware performing operations for Iri and Matsui's system were modified to maintain a register giving the largest exponent-length that has arisen in calculations since this register was last reset under program control, the value in this register could be used to get a lower bound on the lengths of the mantissas arising in these calculations. This bound could be used as the fixed mantissa length is normally used to establish limits on the accuracy of floating-point results.

We believe that the advantages of variable-length exponents outweigh their disadvantages for realistic applications of computer arithmetic, particularly for command and control applications. It is usually more useful to know a number with less precision than to merely know that it overflowed or underflowed. We believe the situation is analogous to that of the partial underflow called for in the IEEE floating-point standard [IEE85], where some of the mantissa bits are discarded to produce nonzero numbers with magnitudes that would otherwise be too small to represent. (We note, however, that partial underflow was one of the most controversial features of the

IEEE standard; c.f., [FW79, Coo81, Dem81].) Variable-length exponents also give greater accuracy for moderately-sized numbers, numbers that arise most often in typical applications.

Iri and Matsui only propose performing arithmetic as if results are first calculated exactly and then rounded to the nearest representable values as in IEEE round-to-nearest rounding. That is the form of rounding used in their simulations of arithmetic operations for their representation. There seems to be no reason, though, why their representation could not be used with each of the four rounding modes possible in IEEE floating-point arithmetic [IEE85].

Iri and Matsui [MI81] give several examples of calculations involving numbers of widely-varying magnitudes, particularly calculations of binomial-distribution probabilities of the form

$$x_k = \binom{N}{k} p^k q^{N-k}$$

for nonnegative integers N and k such that N , k and $N - k$ are all large, and reals p and q such that $p + q = 1$. Simulations of calculations using their system perform significantly better than do either IBM 360 or IEEE-standard floating-point arithmetic. Their arithmetic is also not nearly so vulnerable as the others are to producing inaccurate results because of underflow or being unable to complete calculations because of overflow.

If upward and downward rounding were available in their arithmetic, their representation would be significantly better than floating-point as a means for representing the endpoints of intervals. It does not facilitate parallel computation.

We propose several extensions of Iri and Matsui's representation in Chapter 7, including an extension that does facilitate parallel computation. These extensions use previously described ideas from Matula and Kornerup, and an idea from Aberth [Abe88] to be described in Chapter 6.

5.5 Recurring-Digits Arithmetic

Yoshida [Yos83] proposes a representation system based on extending "recurring decimals" to arbitrary bases and representing them in computer calculations by adjoining "length of the recurrent portion of the mantissa" counts to what would otherwise be ordinary floating-point values. This representation thus provides an alternative way of exactly representing simple rationals as well as ordinary floating-point values.

Generalizing ordinary notation for recurring decimals, for base b the value

$$0.d_1 \cdots d_n \overline{d_{n+1} \cdots d_{n+r}}$$

represents the number

$$\frac{\sum_{i=1}^{n+r} d_i \cdot b^{n+r-i} - \sum_{i=1}^n d_i \cdot b^{n-i}}{b^n \cdot (b^r - 1)}$$

in the interval $[0,1]$. In Yoshida's representation, which she calls FLP/R*, each value contains a "recurring portion" count, a sign, a mantissa and an exponent. On a hypothetical computer with base-10 arithmetic and 10 digits in the mantissas of its floating-point values, for example, the value with recurring portion count 3, positive sign, mantissa 3456756756 and exponent 4 would represent the number $0.34\overline{567} \cdot 10^4$. Because the recurring portion of each value must be on the right end of the mantissa, the mantissa need not be normalized. On the same hypothetical computer, for example, the number $0.1\overline{3} = 2/15$ would be represented by the FLP/R* value with recurring-portion count 1, positive sign, mantissa 0000000013, and exponent 8.

Unlike floating-point, there is duplication in the numbers represented by the mantissas and recurring-portion counts in the FLP/R* system, since in ordinary recurring decimals equalities such as $0.1 = 0.0\overline{9}$ and $0.34\overline{12} = 0.34\overline{1212}$ occur. As might be expected, though, and as Yoshida demonstrates with empirical results, the relative effect of such redundancies on the total number of different numbers representable by FLP/R* values with fixed-size mantissas becomes progressively less as the size of the mantissas increases. Also unlike floating-point, the gaps between representable values with the same exponent in the FLP/R* system are not uniform.

Arithmetic can be performed in the FLP/R* system by expanding out the recurrent portions, if any, of the mantissas of the values being combined to a sufficient number of digits, performing the desired operations on these expanded values as in ordinary floating-point arithmetic, and then rounding the computed values to the nearest values in the FLP/R* system. Yoshida gives the maximum number of digits that each operand's mantissa must be expanded to for each of the arithmetic operations; this number of digits is never more than twice the sum of the number of digits in the operand's mantissa and the length of its recurrent portion.

Since simply incrementing the last digit in a mantissa gives a new value representing a number at least as large as any number represented by the original mantissa with some recurring-portion count, the number of possible FLP/R* values with a fixed number of digits in their mantissas that must be considered to find the one closest to a computed expanded result is at most one larger than this fixed number of digits. As an example, in a base-10 system with 6-digit mantissas the possibilities that must be considered in rounding the expanded result 485656524232, written to make the example clearer as 485656 524232, are just

485656000000,
485656666666,
485656565656,
485656656656,
485656565656,
485656856568,
485656485656, and
485657000000.

The closest of these is the next-to-last one, so in the FLP/R* system the expanded result would be rounded to a value with mantissa 485656 and recurring-portion count 6.

Although she only discusses arithmetic performed as if the results were first computed exactly and then rounded to the nearest representable value, the same arguments that limit the number of possibilities that must be considered in rounding the expanded results also limit the number of possibilities that must be considered in other roundings; her representation could be used for each of the four roundings possible in IEEE arithmetic [IEE85].

Yoshida does not give possible hardware implementations of the FLP/R* system, and does not discuss the critical question of operation speed. She produces her empirical results with an FLP/R* simulator.

Although she does not propose a specific binary implementation of her representation system that would make storage-efficiency comparisons possible, the recurring-portion counts probably cause her representation to be significantly less efficient in using storage than floating-point; contrast this with the use of storage in Hwang and Chang's hybrid floating-point/fixed-slash described in Section 5.3. It would thus not serve as well as a means of representing the endpoints of intervals. The representation also does not facilitate parallel computation.

5.6 Hyper-Exponential Representations

Olver and Clenshaw [Olv87] propose a representation system that is closed under the usual arithmetic operations, so overflow and underflow are not possible. Actually, they propose two systems, one of *level-index* and the other of *symmetric level-index* arithmetic. The level-index system is a simpler special case of the symmetric level-index system. The level-index system is immune from overflow while the symmetric level-index system is immune from both overflow and underflow.

The level-index system uses the "generalized" exponential and logarithm functions, ϕ and ψ , defined for nonnegative real numbers x and X by

$$\phi(x) = \begin{cases} x & \text{if } 0 \leq x < 1, \\ e^{\phi(x-1)} & \text{otherwise;} \end{cases}$$

and

$$\psi(X) = \begin{cases} X & \text{if } 0 \leq X < 1, \\ \psi(\log X) + 1 & \text{otherwise.} \end{cases}$$

The level-index system represents the real X as a sign, a *level*, which is the integer portion of $\psi(|X|)$, and an *index*, which is a fixed-point approximation to the noninteger portion of $\psi(|X|)$.

The symmetric level-index system uses negative index values to denote the reciprocals of values in the level-index system. Its analogs of the generalized

exponential and logarithm functions, Φ and Ψ , are defined for real numbers x and positive real numbers X in terms of the previously-defined functions ϕ and ψ by

$$\Phi(x) = \begin{cases} 1/\phi(1-x) & \text{if } x < 0, \\ \phi(1+x) & \text{otherwise;} \end{cases}$$

and

$$\Psi(X) = \begin{cases} 1 - \psi(1/X) & \text{if } 0 < X < 1, \\ \psi(X) - 1 & \text{otherwise.} \end{cases}$$

The symmetric level-index system represents values as the level-index system represents them, with the extension that the level values are signed integers.

Olver and Clenshaw define methods for performing addition and subtraction in the level-index system by induction on the levels of the values being combined; the induction steps typically involve evaluating powers of e to previously-determined exponents. Multiplication and division can be performed similarly, since taking logarithms or exponentials is equivalent to decrementing or incrementing levels. They have also developed methods for performing arithmetic operations in the symmetric level-index system. They acknowledge that their arithmetic operations can be expected to be significantly slower than the analogous operations in floating-point.

The arithmetic on the symmetric level-index system is closed if the index portions of the values represented are stored in, say, fixed-point binary with a limited number of bits, and if the level portions can be reasonably large. The reason for this closure is that for large values of X the levels of the values $\phi(X)$, $\phi(X + X)$ and $\phi(X \cdot X)$ are equal and their indexes are indistinguishable to the fixed number of bits available to store them. Olver and Clenshaw show, for example, that for 32-bit fixed-point binary indexes this indistinguishability arises for levels less than 6.

For a fixed number of bits, the level-index system is least precise for real numbers $0 \leq X < 1$. The system becomes more precise, temporarily, as X increases, but its precision eventually decays to the point that X and X^2 are indistinguishable.

The precision of the level-index system with 32-bit indexes compares to IEEE floating-point arithmetic as follows: For IEEE single-precision values, the level-index system is up to 32 times more precise for $1 \leq X \leq 2^{11}$, is of roughly the same precision for $2^{11} < X \leq 2^{18}$, and is up to 37 times less

precise for $2^{18} < X < 2^{127}$. For IEEE double-precision values, the level-index system is up to 256 times more precise for $1 \leq X \leq 2^{44}$, is of roughly the same precision for $2^{44} < X \leq 2^{70}$, and is up to 68 times less precise for $2^{70} < X < 2^{1023}$. Beyond these ranges, overflow occurs for the IEEE floating-point values. These precision comparisons are for extreme cases, though, not for typical ones.

Our impression is that this representation gains its advantage of freedom from overflow and underflow with unacceptable losses in computation speed and increased complexity. The variable-length-exponent representation by Iri and Matsui described in Section 5.4 gains comparable immunity from overflow and underflow with much simpler operations. Also, as Demmel notes [Dem87], any representation that allows the precision of stored values to vary makes it more difficult to determine the precision of final results, but a register storing the largest exponent-length used would have a simple interpretation making such a determination possible; a register storing the largest or smallest level-value used would be much less useful, since possible values with a given level have such a great range.

The representation is not appropriate as a means for representing the endpoints of intervals, and it does not facilitate parallel computation.

5.7 Finite-Segment p -adic Representations

Finite-segment p -adic arithmetic, for a prime p , is a method of doing exact rational arithmetic on values that can each be stored in a fixed amount of computer memory. Further, the operations in this arithmetic are similar to those for ordinary floating-point arithmetic carried out in base p .

For the moment, fix p and a positive integer r and let $m = p^r$. Let N be the largest integer such that $2N^2 + 1 \leq m$. Define subsets of the rationals by

$$\hat{Q} = \{a/b : a, b \in \mathbb{Z}, \gcd(a, b) = 1 \text{ and } \gcd(b, p) = 1\}$$

and

$$F_N = \{a/b \in \hat{Q} : |a| \leq N, |b| \leq N\}.$$

If a/b and c/d are rationals in \hat{Q} , call them *equivalent* if $ad \equiv bc \pmod{m}$.

It is then true that for N as above, each equivalence class in $\hat{\mathbf{Q}}$ contains at most one member of \mathbf{F}_N .

Identify the members of the ring $\mathbf{Z}/m\mathbf{Z}$ with the integers 0 to $m - 1$, and identify the operations on this ring with the corresponding operations modulo m . For any integer b such that $\gcd(b, p) = 1$, let b^{-1} be the unique integer k from 1 to $m - 1$ such that $bk \equiv 1 \pmod{m}$; such an integer always exists because $\gcd(b, p) = 1$. Define a mapping ϕ from $\hat{\mathbf{Q}}$ to $\mathbf{Z}/m\mathbf{Z}$ by letting ϕ map the rational a/b to the modulo- m product of a and b^{-1} . This mapping is a ring homomorphism.

The process of using finite-segment p -adic representations works as follows: Start with rational numbers for which some computation is desired, and choose p so that all these rationals are in $\hat{\mathbf{Q}}$. Choose r sufficiently large so that for m and N as above, all these rationals are in \mathbf{F}_N . Apply the mapping ϕ to the rationals, and perform all the desired sums, products and so on, as the corresponding operations in $\mathbf{Z}/m\mathbf{Z}$ using *finite Hensel codes*, which are finite segments of real numbers' representations as p -adic values. (Finite Hensel codes, algorithms defining the mapping ϕ and its inverse, and algorithms defining the arithmetic operations in $\mathbf{Z}/m\mathbf{Z}$ in terms of these codes are described completely in [GK84]; the codes, the operations on them, and the advantages and disadvantages of using them are described informally in the following paragraphs.) If the result of these computations is the image under ϕ of a member of \mathbf{F}_N , this rational is the exact result of the desired computation on the original rational numbers.

The finite Hensel codes for particular values of p and r as above are like ordinary base- p floating-point values with r -digit mantissas. The arithmetic operations on these finite Hensel codes are similar to those of ordinary base- p floating-point arithmetic, with the exception that carries are carried to the *right* rather than the left; this corresponds to the condition that two reals are close in the p -adic metric if and only if their difference is a rational whose reduced form has a numerator divisible by a *large* power of p . As a consequence, the successive digits in a finite Hensel code result can be calculated from left to right, one at a time, with no possibility that later calculations will change these digits.

Unfortunately, \mathbf{F}_N is not closed under the operations of addition, subtraction, multiplication and division. This gives rise to the phenomenon called

pseudo-overflow, which makes it impossible to associate a unique member of F_N with the result of a finite-segment p -adic calculation. In such a situation, all that is known about the desired exact rational answer is its equivalence class in \hat{Q} . If machine integers of a fixed size are used to represent the powers of p in the finite Hensel codes, ordinary integer overflow is also possible.

If pseudo-overflow occurs, it is possible to simply repeat the calculations with a larger value of r . Since the computations of earlier digits are never changed by the computations of later ones, no results already computed have to be recomputed. Increasing r until pseudo-overflow does not occur is thus analogous to demanding more digits or more continued-fraction partial quotients until a desired degree of accuracy is obtained. One could also do similar calculations with another prime p' and another power s . Since powers of p and p' are relatively prime, the separate calculations with p and r and with p' and s are accurate for all the order- N' Farey fractions for any N' such that $2(N')^2 + 1 \leq p^r \cdot (p')^s$. The two calculations together are more likely to result in pairs of Hensel codes that have a unique interpretation.

Finite-segment p -adic arithmetic provides an efficient way of performing exact rational arithmetic when pseudo-overflow does not occur. It also gives an efficient way of representing rationals, so that for reasonably small values of p and r the family of representable rationals includes most rationals that are likely to arise in computations.

Finite-segment p -adic arithmetic has the major defect, though, that it does not provide a natural means for discarding information when pseudo-overflow occurs. The p -adic metric is drastically different from the normal one, so each rational in \hat{Q} has both rationals very close to it and rationals very far away from it in its equivalence class.

This representation is not appropriate as a means for representing the endpoints of intervals, and does not facilitate parallel computation.

Chapter 6

Constructive Reals

Several of the representation systems that we discussed in our interim report [ORA88] were based on the constructive reals. This chapter describes our work in this area. The chapter begins with a general definition of the constructive reals and brief comments on their properties. It describes basic advantages and disadvantages of constructive-real arithmetic, and lists implementations of this arithmetic, including implementations based on continued fractions and convergent sequences of rationals. The chapter then presents the results of our work with continued fractions and gives more information on Boehm's implementation of constructive-real arithmetic, an implementation based on convergent sequences of rationals.

6.1 Definitions and Basic Properties

A *constructive*, or *recursive*, real is a real number for which there exists a finite algorithm capable of generating arbitrarily-accurate rational approximations to this real. We will follow Boehm [Boe87] in calling these reals *constructive*, though they have been called *recursive* by recursion theorists [Rog67]. The constructive reals include all rational numbers, all algebraic numbers, and an infinite number of transcendental numbers, including e and π . Intuitively, every real number for which a method exists for computing that number arbitrarily accurately is a constructive real, so if the ideal inputs to ideal

computer calculations were known arbitrarily accurately then all the numbers arising in these calculations would be constructive reals.

There do not exist valid general algorithms for deciding whether two constructive reals are equal or whether one is larger than the other, or for deciding whether a constructive real is 0 or a rational [Rog67]. Ordinary floating-point arithmetic does "better" in being able to decide equality and order, and in being able to determine that a number is 0, only because it represents only a finite number of reals. If two constructive reals are considered equivalent if they are closer together than a user-supplied tolerance, then equality, order, and being different from 0 are all decidable for the constructive reals.

Systems implementing constructive-real arithmetic take algorithms computing particular constructive reals and create, under user control, algorithms for computing sums, differences, products, quotients, logarithms, exponentials, etc., of these reals. The user inputs reals that the systems take to be exact, and they execute the algorithms they create for producing sums, products, quotients, etc., to produce numerical results to user-specified degrees of precision. In actual implementations, of course, the amount of precision obtainable is limited by available computer memory and the time needed to compute the results. In Aberth's [Abe88] implementation of an arithmetic that allows the user to set the precision, for example, at most roughly 120 decimal digits of precision are available.

The constructive reals are strongly related to the notion of *on-demand* or *data-driven* precision. In a system providing on-demand precision, the user specifies a tolerance for relative error in the final results of a calculation. The system then carries out the intermediate calculations to *whatever* degree of precision is necessary to give final results that are guaranteed to be as precise as the user has specified. The degree of precision necessary for intermediate results can vary with the numbers that actually arise; in evaluating $1/(1-x)$, for example, x must be evaluated more precisely if it is approximately equal to 1.

6.2 Advantages and Disadvantages

The basic advantage of implementations of constructive-real arithmetic is that they produce results that are practically arbitrarily accurate. These results do not depend on anomalies of a particular form of finite arithmetic, such as floating-point, and are never misleading because information that later turned out to be significant was thrown away.

Systems providing “on-demand” precision can be useful even if there are upper limits on the amount of precision the user can specify for final results or that the system can take for intermediate ones. Such systems do not perform constructive-real arithmetic, but can be useful in situations where the amount of precision needed is variable but typically low; they can save computation time by not calculating any intermediate result more precisely than they have to.

We noted in our interim report, though, that since constructive-real systems do not discard information the amounts of time and space they use can easily become excessive. In situations where high precision is needed, the systems can fail to provide this precision within the available space and time. In more typical situations where high precision is not needed, the systems’ memory-management overhead for dynamically making space available for results can slow things down significantly [KL85]. More importantly, in calculations with large numbers of intermediate results, particularly calculations involving loops, it can be impossible for these systems to store all the *algorithms* they create for generating arbitrarily-accurate approximations to these intermediate results.

In calculations involving loops, a constructive-real system can compute intervals containing intermediate results to a high degree of accuracy, then recompute these intervals to a higher degree of accuracy if they do not give final intervals sufficiently short to guarantee the user-specified degree of precision. (C.f., Aberth [Abe88].) Such a process can be expected to be slow, though, and can easily be unacceptable in situations requiring real-time results.

On the issue of possibly using constructive-real systems for real-time applications, Hans-Juergen Boehm, one of the principal developers of the constructive-real system described in Sections 6.3 and 6.5, advised us [Boe89]:

My view is that none of this is presently suitable for real-time applications. It is probably too slow at present, and too hard to guarantee response time.

Boehm raises the critical point that if additional computation is done in a data-dependent fashion to maintain a desired degree of precision, this introduces a possibly unacceptable dependence of the *response times* on the input data.

In our interim report, we planned to seek practical situations where only a limited amount of precision is needed in results that can be taken as final, even if they are used as inputs to further calculations, but where the precision needed in intermediate results to obtain this final precision is unpredictable. Finding the determinant of a large matrix to a moderate degree of precision is an example, since the matrix can be singular or nearly singular. We speculated that constructive-real systems might be useful in such situations, even for real-time applications.

We have since learned, though, that having intermediate results be more precise than initial inputs is generally only useful for avoiding the introduction of calculation error in computations with large numbers of intermediate results. The determinant of a nearly-singular matrix shows the problem: Although any matrix has an exact determinant, computing this determinant to a high precision implicitly assumes that the inputs defining the matrix are exact, an assumption that is usually not warranted. A computation with a large number of intermediate results is exactly the sort of situation where a constructive-real system can be expected to perform poorly.

Other situations that require unpredictable precision in their intermediate results also require unpredictable precision in their inputs, so are unrealistic for real-time applications. Techniques for representing constructive real numbers and performing operations on them are thus more likely to be useful for real-time applications as ways of providing "on-demand" accuracy to reduce the necessary amount of computation.

6.3 Implementations

The systems doing constructive-real arithmetic considered in our interim report included ones representing real numbers as lazily-evaluated lists of redundant digits [Pix82] or of partial quotients [Jon84]. In *lazy evaluation*, a potentially-infinite list is represented as an initial segment of that list together with an algorithm capable of extending the list on demand [ASS85]. As in Section 3.4, redundant digits avoid the impossibility, say, of not knowing whether to output 2 or 3 as the third digit of an ordinary base-10 number whose calculation begins 8.72999.... A list of partial quotients can give either a number's standard continued fraction or one of its generalized continued fractions. As we mentioned in Section 4.5, and will discuss in Subsection 6.4.2 below, generalized continued fractions give exactly the same sorts of advantages that redundant digit-sets give.

Our interim report also considered an implementation of constructive-real arithmetic by Boehm [Boe87] that represents a real x as a function f_x from the integers, where the integer input specifies the required precision, to the rationals, where the rational returned approximates x to the required precision. For efficiency, this system represents rationals as appropriately-scaled integers, uses interval arithmetic in some of its intermediate calculations, and records the results of earlier computations so that when it is computing a new approximation to a real it always has the most precise previously calculated approximation to that real available. We give additional information on Boehm's system in Section 6.5.

Jones [Jon84] has implemented a version of Gosper's algorithm in SASL [Tur76], a functional programming language strongly and historically related to the Caliban language used in the Clio prover [BMS89], the theorem prover used by the Reals project. SASL treats every object as a function, so functions can be passed as arguments to functions and returned as values of functions. This makes it natural for implementing a version of Gosper's algorithm that takes functions — i.e., infinite lists of partial quotients — as arguments and returns them as values.

Boehm's system is implemented in Russell [BDD80], a strongly-typed functional language. Like SASL, Russell treats functions as first-class objects that can be passed as arguments to, and returned as values by, functions. It

is also natural for implementing an arithmetic that treats real numbers as functions — in this case, functions from integers specifying desired precisions to integers specifying sufficiently-precise rational approximations.

In principle, the LCF representation, particularly with the extensions to redundant bit-sets mentioned in Subsection 5.2.4, could be used in a system of constructive-real arithmetic. Matula and Kornerup's primary interest in this direction, though, seems to be in using LCF in "on-demand precision" arithmetic systems that minimize necessary computation.

Chapter 8 lists references to iterated-interval arithmetic systems. These also implement constructive-real arithmetic or "on-demand precision" approximations to it.

6.4 Continued Fraction Results

We implemented code for generating standard and optimal continued fractions in both Caliban and C. We also implemented C code carrying out Gosper's algorithm to take standard or generalized continued fractions as inputs and produce standard or approximately-optimal generalized continued fractions as outputs.

6.4.1 Caliban Continued Fractions

Appendix H contains the Caliban program we implemented for finding the standard and optimal continued fractions for rationals selected by user input. The user executes this program in the Clio prover by simplifying the expression `getcfrac n`, where n is a positive integer less than 134217728, which is the maximum value of one of Caliban's NUM's. (If the user wants the program to terminate in reasonably short order, he or she should use a much smaller value of n .) The Clio prover simplifies the expression to a list containing all the fractions i/n for $n \leq i \leq 2n$, given as numerator/denominator pairs, and their standard and optimal continued fraction expansions.

Every operation used in this program is completely defined within Appendix H except for the arithmetic operations on NUMs. Definitions of the natural numbers, the integers, the rationals, the usual arithmetic opera-

tions on these numbers, the order and equality relations on these numbers, and the absolute-value function on these numbers, are contained in the file `arith.def`, which is included in Appendix H.

We were originally interested in using Caliban to implement Gosper's algorithm because we had already obtained Jones' [Jon84] implementation of this algorithm in SASL. A language that treats functions as first-class objects is also natural for constructive-real arithmetic. The process of simplifying Caliban expressions is much slower than the process of executing C code, though, and Caliban does not contain the convenient I/O statements available in C, so we decided to use C rather than Caliban for our further investigations into continued fractions and Gosper's algorithm.

We implemented a C program carrying out the same sorts of calculations as those performed by the Caliban program in Appendix H, and in addition suppressing output for fractions whose standard and optimal continued fraction expansions are the same, computing convergents for both standard and optimal continued fractions, and maintaining counts of the numbers of partial quotients in both forms of continued fractions. This program led us to observe some of the properties of continued fractions and their convergents noted in Chapter 4, but we have not included it because we do not believe it is of further interest.

6.4.2 Gosper's Algorithm

Sections 1 and 2 of Appendix I contain our C implementations of Gosper's algorithm for standard and generalized continued fractions. These programs only compute combinations of two specific infinite continued fractions, and must be modified and recompiled to compute combinations of other continued fractions. As they are given, the programs only compute combinations of $1 + \sqrt{2}$, which has a particularly simple standard continued fraction expansion that is also optimal, with itself:

$$1 + \sqrt{2} = [2, 2, 2, 2, \dots] \approx 2.414213562373095.$$

The parts of these programs that must be changed to compute combinations of other continued fractions are very specific, though: The functions `x` and `y` correspond to the inputs x and y of Gosper's algorithm. These

functions take no arguments; the i th time x or y is called it returns the i th partial quotient of x or y . When the programs x and y perform in this way, the C code implementing Gosper's algorithm is similar to functional programming language code that treats x and y as lazily-evaluated infinite lists. The functions rx and ry take no arguments and return double-precision approximations to x and y , respectively; they are used for producing descriptive output.

An easy way to have different calls to a C function with no arguments return different values is to have a static counter local to the function distinguish the calls. Section 3 of Appendix I contains such a function that computes the standard continued fraction for e , the base of the natural logarithms.

Both programs use double-precision values to store the entries of the coefficient cube for two reasons:

1. Double-precision floating-point values, at least on most computers, can exactly represent larger integers than integer values can; and
2. In IEEE arithmetic, the hardware maintains a status flag that records whether the results of floating-point computations are exact or approximate, making it possible to detect when coefficient-cube updates are approximate. (Aberth [Abe88] seemed to not be aware of this possibility, but he was working on IBM machines on which it does not exist.)

The programs take the partial quotients of the inputs and the output to be in double-precision in order to be consistent with their handling of the coefficient cube.

Both programs assume that their inputs are infinite continued fractions, and make no attempt to handle inputs that can be exhausted. They could presumably be rewritten easily to test for the IEEE $+\infty$ value as a partial quotient and use it to mark the end of an input — c.f., the discussion of treating finite continued fractions as having $+\infty$ as their final partial quotient in Subsection 5.2.1. Instead, both programs ingest partial quotients of their inputs and output partial quotients of their outputs for as long as their coefficient-cube updates are exact.

Both programs use floating-point arithmetic to decide whether output is possible or from which of x or y to ingest another partial quotient; since this arithmetic is approximate, their decisions can be incorrect, particularly after enough terms have been ingested to make the distinctions between the alternatives small. Neither program attempts to determine whether its output should terminate, but this is appropriate since it is not generally possible to determine whether a constructive real is rational.

The standard continued fraction program implements Matula and Kernerup's [KM88] version of Gosper's algorithm. In particular, it maintains a decision cube to give the four extreme values $z(1,1)$, $z(1,\infty)$, $z(\infty,1)$ and $z(\infty,\infty)$, and ingests a partial quotient from x if the integer parts of its floating-point estimates of $z(1,1)$ and $z(\infty,1)$ are not equal. If these two integer parts are equal, it ingests a partial quotient from y if the integer part of either of its estimates of $z(1,\infty)$ or $z(\infty,\infty)$ differs from the other three integer parts. If all four integer parts are equal, it outputs the common integer part as the next partial quotient of z .

The generalized continued fraction program does not use a decision cube and does not attempt to minimize the computations it performs to decide whether to produce output or how to ingest another partial quotient. It ingests a partial quotient from whichever of inputs x or y causes the greatest variation in its directly-computed floating-point estimates of $z(x,y)$ when that input is replaced by the extreme values of $-\infty, -1, 1, +\infty$ and the other input is held constant at one of these four extremes. Actually, it considers only nine cases instead of sixteen, because $z(-\infty, y) = z(+\infty, y)$ and $z(x, -\infty) = z(x, +\infty)$. The program outputs a partial quotient of z when all its estimates of z at the extremes of x and y are within $1/2$ of each other; if it outputs a partial quotient, it takes this partial quotient to be the integer nearest the average of the largest and smallest of its estimates of z at these extremes. The program assumes that the possible values of z are bounded by its values at the extremes of x and y ; we did not check this assumption carefully, even for initializations of the coefficient cube that perform the four basic operations of arithmetic.

If it were not for possible error in the estimates of z , having the extreme values of z be within 1 of each other would guarantee that there is an integer that is within 1 of all the possible values of z . By using the tighter bound of

$1/2$, and by taking the partial quotient it outputs to be the integer nearest the average of the largest and smallest extreme values, the program chooses a partial quotient that is within 1 of all the extremes of z even with error in its estimates of these extremes. Further, this partial quotient will typically be within $1/2$ of all the possible values of z .

The program thus computes an approximation to an optimal continued fraction. The program only assumes that its inputs are generalized continued fractions; if it assumed its inputs were optimal, it could evaluate values of z at the extremes $\pm\infty$ and ± 2 instead of $\pm\infty$ and ± 1 .

Even though the inputs x and y are fixed, the user can compute many different combinations of these inputs by varying the initial entries of the coefficient cube. The following remarks are based on computations for $x = y = 1 + \sqrt{2}$ of $x \cdot y$, $x + y$, $x - y$ and x/y .

When the result is rational, as in $x - y = 0$ or $x/y = 1$, the standard continued fraction program "hangs", falling victim to the main "catch" in Gosper's algorithm. It outputs no partial quotients at all. The generalized continued fraction program produces the correct first partial quotients of 0 or 1, then produces no more partial quotients until it terminates because of an inaccurate coefficient-cube update. After it outputs the first partial quotient, giving a perfectly correct "approximation" to the final result, the possible extreme values of z computed by the generalized continued fraction program vary over *larger and larger* extremes as the program ingests more partial quotients from x and y . This is reasonable, since either $+\infty$ or $-\infty$ as the next partial quotient would be correct.

For $x \cdot y = 3 + 2\sqrt{2}$, the standard continued fraction program outputs $[5, 1, 4, 1, 4, 1, 4, \dots]$ while the general continued fraction program outputs $[6, -6, 6, -6, 6, -6, \dots]$. The general continued fraction program's output converges, on a partial quotient per partial quotient basis, to its ideal value more quickly than does the standard continued fraction program's output, but otherwise their results are similar. At the point that floating-point approximations to the convergents of both outputs and to the ideal answer become identical, the standard and generalized continued fractions contain 21 and 11 partial quotients, respectively; the standard continued fraction has a pair of partial quotients 1 and 4 for every generalized continued fraction partial quotient of 6 or -6. The outputs for $x + y = 2 + 2\sqrt{2}$ are similar.

These results were contrary, at least in practical terms, to our expectation that the more-uniform throughput possible for generalized continued fractions would reduce the magnitudes of the coefficient-cube entries arising in the calculations. Coefficient-cube entries with similar magnitudes are needed by the two programs to produce outputs with a given precision. In the $x \cdot y = 3 + \sqrt{2}$ case, the largest magnitudes of the coefficient-cube entries for the two programs at the times when floating-point approximations to the convergents first equal floating-point approximations to the ideal answers are identical. In particular, in both programs the coefficient-cube entries typically grow without bound as the programs ingest and output more and more partial quotients.

The numbers of partial quotients the two programs must ingest to produce outputs with a given precision are roughly equal. The generalized continued fraction program must occasionally ingest a few more partial quotients because it must consider more possibilities in bounding the values of z .

It is noteworthy, though, that for both programs the partial quotients converge to the best possible value representable in double-precision long before the coefficient-cube entries become too large to represent exactly in double-precision. This is presumably related to why the Matula and Ferguson [FM85] results described in Subsection 5.1.4 on using simulated mediant rounding to find the inverses of Hilbert matrices are so much better than the floating-point results for these same calculations, even though the simulation uses exactly the same floating-point hardware.

6.5 Boehm's Constructive Reals

Boehm's [Boe87] constructive-real system comes in an arbitrary-precision "desk calculator" that runs on Sun workstations. The calculator is able to produce results whose precisions are limited only by available computer memory and user patience. The amount of patience it requires is very reasonable; on a Sun 3/60, we used the calculator to compute e^e to 1028 decimal places in less than 149 seconds.

We originally planned to test the time and space requirements of Boehm's system on a problem with a large number of intermediate results by finding

the determinant of a large matrix, but we were unable to do so before we were asked to finish Task 5. Such a calculation is difficult to do with the calculator because the calculator only performs operations in response to user input; it is not programmable. The following paragraphs summarize the information we gathered showing how Boehm's system could be used for the sort of problem we intended to test it on, and give recent results that Boehm gave us on the system's speed. We have already noted Boehm's opinion on the underlying issue of whether constructive-real representations might be useful for real-time command and control applications.

Boehm and Vernon Lee have developed a matrix-arithmetic package written in Russell that can be used with the constructive-real package. It does not include taking determinants, but does include a Gaussian elimination routine that could be used as an example of a problem of similar complexity. In addition, recent versions of the Russell compiler include source for versions of the constructive-real package that can be called from C code.

Boehm gave timing results for problems requiring on the order of 100 intermediate results in [Boe87]. For example, on a Sun 3/260 with a Motorola 68881 floating-point coprocessor, with sufficient heap space already allocated in computer memory, the constructive-real package took about 183 seconds to compute $100!$ by taking the natural logarithm of each of the numbers from 1 to 100, adding these logarithms, and finding the exponential of the result. Current speed numbers for the package [Boe89] are about 20% better than the results reported in [Boe87] because of incremental improvements to the package and the Russell compiler.

Boehm seems to have gotten a factor of 10 speed improvement with newer implementations of the package based on iterated interval arithmetic, but these implementations are not yet ready to be distributed [Boe89]. We give brief comments about iterated interval arithmetic in Chapter 8.

Chapter 7

Conclusions and Questions

This chapter summarizes our conclusions and lists possible questions for future research. The conclusions are primarily addressed to Task 5's Air Force sponsors.

7.1 Conclusions

Although the absolute bounds on error given by interval analysis are desirable, the bounds given by simply replacing scalar arithmetic operations with corresponding interval ones are so overly conservative that they usually do not correctly show the dependence of computed outputs on errors in inputs and intermediate calculations. Matijasevich [Mat85] has suggested a technique for efficiently computing partial derivatives that might lead to a way of limiting this problem, but his technique does not yet adequately handle programs with loops. Simply replacing scalar arithmetic operations with corresponding interval ones also does not give a useful extension of the Real's project's notion of asymptotic correctness. More sophisticated interval algorithms can produce surprisingly perfect results, though, and should be investigated for their possible command and control applications.

The trade-offs involved in making the asymptotic model more and more realistic are difficult to evaluate since the time, space and circuit-complexity costs of increasing the precision of floating-point arithmetic are high. How-

ever, the asymptotic model is presumably reasonably accurate, since these trade-offs were implicitly considered in designing recent forms of floating-point arithmetic. Adherence to the IEEE floating-point standard seems to force some operations, particularly division, to be performed more slowly, and IEEE arithmetic is slightly less precise than VAX arithmetic, for double-precision values, because it allocates more bits to the exponent. Versions of floating-point arithmetic with redundant digit sets facilitate parallel processing by supporting on-line arithmetic.

There are alternative representation systems in the engineering literature that are good for situations involving rational numbers and/or highly parallel computations. Command and control applications should be examined for the presence of such situations to identify cases where these representation systems might be useful. There is also one alternative representation system, the variable-length-exponent version of floating-point by Iri and Matsui [MI81], that we believe is generally preferable to the standard floating-point representations for command and control applications.

Approximate rational arithmetic provides a means for capturing many of the advantages of exact rational arithmetic without incurring unacceptable time and space costs. Mediant rounding, with its bias towards simplicity, can greatly increase the accuracy of computations on numbers whose ideal values are rational. Representations based on continued fractions, particularly redundant generalized continued fractions, treat exact rational and approximate real computations uniformly, and facilitate parallel processing by supporting arithmetic operations based on Gosper's algorithm that can be carried out on-line and largely in parallel.

The most mathematically interesting representation systems in the literature are the extended floating-slash [MK85] and binary lexicographic continued fraction (LCF) [KM88] representations by Matula and Kornerup. These systems implement approximate rational arithmetic, and experimental results indicate that floating-slash is much more accurate than floating-point for computations on quantities whose ideal values are rational and is not significantly worse than floating-point on other computations. In addition, the extended floating-slash representation has the same ability as floating-point to conveniently represent numbers of widely varying magnitudes. The LCF representation makes it easy to decide which of two numbers is larger,

and also supports bitwise, on-line, variable-accuracy arithmetic. The LCF representation does not have floating-point's ability to conveniently represent numbers of widely varying magnitudes, but a possible extension of this representation might.

Approximate rational arithmetic is probably most useful in command and control applications for knowledge-based systems in which it is critical to recognize simple rationals. It might also be useful in combinatorial optimization problems involving integer quantities. In situations where it might be useful, simulating median rounding in floating-point might suffice to capture most of approximate rational arithmetic's advantages, and would do so with standard hardware.

The alternative representation system in the literature that we believe would be most useful for typical command and control applications is the variable-length-exponent representation by Iri and Matsui [MI81]. This system is capable of representing quantities of such widely-varying magnitudes that it practically eliminates overflow and underflow, and it is more accurate than floating-point for typical, moderately-sized numbers. We believe that this system should be implemented, as Demmel [Dem87] suggests, with hardware recording the maximum length of an exponent that has arisen in calculations. We believe Iri and Matsui's system, with the extension suggested by Demming, is preferable to versions of floating-point that use comparable numbers of bits.

Other alternatives in the engineering literature, particularly Hwang and Chang's [HC78], combine the advantages of floating-point and approximate-rational representations or give other methods of avoiding overflow and underflow. We do not believe these representations would be as effective for these purposes as the floating-slash and variable-length-exponent representations.

Representations based on the constructive reals, in which quantities can be calculated to a virtually-arbitrary user-set precision, are unlikely to be useful for real-time command and control applications. Implementations using these representations tend to be slow, and they pay for predictable precision with unpredictable response times. Further, implementations based on such representations typically assume that inputs are exact, which is unrealistic for command and control applications. However, these representations

are very useful for interactive analysis, and they suggest the desirability of representations that support variable-precision arithmetic.

We propose several possible representation systems based on ideas from Iri, Matsui, Matula, Kornerup and Aberth in Section 7.2. These representation systems raise questions about continued fractions that we give in Section 7.3, and the continued fraction questions raise an open mathematical question about possible representations of integers that we give in Section 7.4.

7.2 Variable-Length-Exponents

This section lists several possible variants of Iri and Matsui's [MI81] variable-length-exponent representation. All these variants are based on using self-delimiting exponents, as in Matula and Kornerup's LCF encoding of positive integers [MK83], rather than a fixed-length field giving exponent lengths. Some of these variants also perform approximate rational arithmetic, support variable-precision arithmetic, or facilitate parallel processing.

Our primary interest in Iri and Matsui's variable-length-exponent representation is not in its immunity from overflow and underflow, but in its efficient use of bits. It first takes bits for the exponent, which gives the number's general magnitude, and leaves any remaining bits for the mantissa. We will initially assume that the exponent is for base 2, as it is in [MI81], and we will ignore negative exponents — it would be easy enough to code numbers having negative exponents by giving each number two sign bits, one for the number and another for its exponent. A number's exponent gives the most significant information about the number, so it is appropriate that as many as necessary of whatever bits are available for coding the number be used to code its exponent.

If the exponents in quantities that arise most often are small, reserving 6 bits to give the length of the exponent, as Iri and Matsui do in [MI81], wastes bits. In the LCF encoding of positive integers, for example, the integers 1, 2, 3 and 4 are coded by the strings 0, 100, 101 and 11000, respectively. The LCF encoding only has 6 “wasted” bits — the leading 1's that give the length of the integer's binary value after that value's leading 1 — for integers greater than 63. Floating-point numbers roughly as large as $2^{64} \approx 10^{19}$ are rare, so

using an LCF encoding of the exponent would typically leave more bits free to be in the mantissa.

Further, the coded value of an exponent could be shifted by a constant to minimize the typical lengths of exponents' LCF encodings. If statistical studies indicated that the binary exponent occurring most often in typical calculations was 3, for example, a number's exponent could be taken to be an LCF exponent value plus 2.

Also, with self-delimiting exponents the lengths of the bit strings used to represent numbers need not be fixed. The initial bits of one of these strings could define a self-delimited exponent and any remaining bits could then be the mantissa. The precision to which results were calculated could then be controlled by a bit- or byte-count number-length value maintained in hardware and subject to program control; Aberth's [Abe88] PRECISION variable serves a similar purpose. Special "exponent" values, such as number fields containing only 1's, could be used to code infinite and not-a-number values for each precision, as well as ± 0 .

As we noted, Iri and Matsui [MI81] only propose performing arithmetic as if results were first calculated exactly and then rounded to the nearest representable values. There seems to be no reason, though, why their representation could not be used with each of the four rounding modes possible in IEEE floating-point arithmetic [IEE85]. We would do so, and would also adopt Demmel's [Dem87] suggestion of maintaining a hardware register to record the largest exponent-length that has arisen in calculations for the current precision.

We have so far assumed that exponents are for the base 2 and mantissas are ordinary base-2 values. It would also be possible, though, to use continued fractions as mantissas and use LCF encodings of these mantissas. That would make the system able to exactly represent a great many rationals, with the number of exactly-representable rationals determined by the current precision. (Every rational has a finite continued fraction, and a rational is exactly representable for a given precision if this precision is large enough to code that continued fraction.) This would also presumably simplify the hardware needed to perform arithmetical operations on quantities of varying precisions (i.e., bit lengths) and increase the numbers of operations that could be done in parallel. If this were done, some base other than 2 might be better

for the exponent, though our initial guess is that it would not be. If this were done, it might also be useful to allow mediant rounding as a rounding option. It might also be useful to allow some sort of redundant-binary LCF-like encoding of the mantissas, as suggested by Matula and Kornerup [KM88].

This last proposal is essentially the same as Matula and Kornerup's [KM88] LCF proposal, including its possible extension to redundantly-coded binary values. It adds a self-delimited initial exponent, an explicit mention of precision bounds, a largest-exponent-length register, and more possible roundings.

Finally, it might be useful to code mantissas as binary representations of generalized continued fractions rather than as redundant-binary representations of standard continued fractions. The issue is whether the more rapid convergence of optimal or nearly-optimal generalized continued fractions, with their more uniform throughput for Gosper's algorithm, is worth the cost of the bits needed to give the signs of the partial quotients. This issue is behind some of the questions about continued fractions in Section 7.3.

The main potential difficulties we see with these possible representations are with operation speed and the size and complexity of the necessary hardware. These issues also arise in the questions in Section 7.3, and are behind the theoretical question in Section 7.4.

7.3 Continued Fraction Questions

This section lists questions about using generalized continued fractions and about using Gosper's algorithm to do arithmetic on them.

Would it be practical to use Gosper's algorithm, and initialize the coefficient cube's entries appropriately, to get the effect of multiplying standard or generalized continued fractions by powers of a fixed base? If not, it would not be practical to give numbers as exponents and continued-fraction mantissas.

What, if any, relationship is there between using redundant-binary arithmetic to code the LCF results of having Gosper's algorithm output standard continued fractions, as Matula and Kornerup [KM88] suggest, and using

signed partial quotients to code the results of having Gosper's algorithm output generalized continued fractions? Would generalized continued fractions give outputs that are easier for hardware to interpret?

Which values of z in Gosper's algorithm must be evaluated for generalized continued fraction inputs fractions in order to definitely bound all possible values of z ? Is it possible to maintain a generalized "decision cube" to give these bounds?

How rapidly can the entries of the coefficient cube grow for redundant-binary output, either for standard or generalized continued fractions?

7.4 Integer Representations

Our study of Gosper's algorithm, particularly the possibility suggested by Matula and Korerup [KM88] that a redundant-binary encoding of the algorithm's outputs might improve its throughput, led us to wonder whether there might exist an encoding of the partial quotients of standard or generalized continued fractions that would allow Gosper's algorithm to produce an indefinite number of partial quotients even if the magnitudes of its coefficient-cube entries were bounded. Thinking about a special case of this, the case in which only one partial quotient is ingested from each of the two inputs, led us to pose the following question about possible representations of the integers:

Informally, the question is whether there exists a (presumably redundant) encoding of the integers that makes it possible for finite-state machines to compute sums and products and also to decide order. Let A^* be the set of all finite strings of elements of an alphabet A . To ask the question about integer encodings precisely, do there exist the following:

- A finite alphabet A ;
- A recursive function f from A^* onto the integers;
- Two restricted Turing machines M_1 and M_2 , both with two one-way input tapes and one one-way output tape, which compute the functions $g_1, g_2 : A^* \times A^* \rightarrow A^*$ respectively; and

- A restricted Turing machine M_3 with two one-way input tapes that always halts when started with elements of A^* on its input tapes and always halts in an accepting or rejecting state;

having the properties that for all $x, y \in A^*$,

1. $f(g_1(x, y)) = f(x) + f(y)$,
2. $f(g_2(x, y)) = f(x) \cdot f(y)$, and
3. M_3 accepts (x, y) if and only if $f(x) \leq f(y)$?

Note that the condition required to give an affirmative answer to this question is very strong. A *single* machine must be able to multiply arbitrarily large integers if they are coded appropriately. It does not suffice to have a single machine *unit* that can be composed into arrays such that any pair of integers can be multiplied by one of these arrays, as is done in [Atr65].

When we first posed a variant of this question, we presumed that the answer was “no”, and expected to go on to consider how rapidly the number of states the coefficient-cube entries can assume increases as the total number of partial quotients Gosper’s algorithm ingests and outputs increases. We learned, though, that this seemingly simple question about encoding the integers is more subtle than we had imagined, and is still open. Professors Hopcroft, Hartmanis and Kozen of Cornell University advised us that they knew of no work that answered this question, and in a recent paper Regan [Reg88] mentioned a ring-theoretic generalization of the question as being open.

For the remainder of this section, we will take “finitely computable” to mean “computable (for a function) or decidable (for a relation) via a fixed recursive encoding by a restricted Turing machine with one-way input and output tapes”. With this terminology, the question we have asked is, “Is there a possibly-redundant recursive encoding of the integers such that addition, multiplication and order are all simultaneously finitely computable?” Replacing “order” by “equality” gives a simple weakening of the desired conditions — i.e., M_3 accepts (x, y) if and only if $f(x) = f(y)$. We do not require that the encoding f itself be finitely computable, but only that it be recursive, so we do not impose any time or space restraints on the computations needed, say, to convert strings in A^* to ordinary signed base-10 numbers.

The answer to our question is "yes" if it is weakened:

- A signed variant of unary notation with a symbol for 0, where the positive integer n is represented by n consecutive 1's, makes addition and order finitely computable.
- A signed variant of prime-power notation with a symbol for 0, where the positive integer n is represented by a string of positive integers in unary that give the exponents of n 's factorization into primes, makes multiplication and equality finitely computable. For example, if commas separate the exponents and # marks the end of the number, since the consecutive primes are 2, 3, 5, 7, ... the encoding of $84 = 2^2 \cdot 3 \cdot 7$ is 11,1,0,1#.
- If prefix-Polish expressions in ordinary base-10 notation are taken to be encodings of the numbers obtained by evaluating these expressions, then addition and multiplication are finitely computable. If + and * denote addition and multiplication respectively, and if commas separate different expressions, the product of 857 and 973 can be "encoded" as *857,973.

Compositions of finitely-computable functions need not be finitely computable, and the answer to our question is more dependent on exactly how it is formulated than one might expect. For example, even though letters from a finite alphabet on two separate input tapes can be coded by a letter from a larger alphabet on a single input tape, our question cannot necessarily be adapted to machines with a single input tape. The question, "Are there twice as many x's on tape 1 as there are on tape 2?" is easily answerable by a restricted Turing machine with two separate, independently-controllable one-way input tapes, but is not answerable by such a machine if the two input tapes are both coded onto a single one-way tape.

With a redundant system, a steady output stream of digits also does not necessarily mean a steady output stream of useful information: 1111111 = 0000001. Still, with generalized continued fractions and Gosper's algorithm it is possible to steadily output partial quotients that all give significant information about the true value of the output, so this suggests that an encoding of the integers making addition, multiplication and order all finitely

computable might have the additional property that each letter of a word imposes significant bounds on the integer coded by that word.

We still expect the answer to our question to be "no", but if there is an encoding of the integers that makes the answer "yes", and if with this encoding the necessary compositions of finitely-computable functions needed to carry out Gosper's algorithm are themselves finitely computable, then there exists a representation system based on continued fractions for which a fixed hardware unit can perform operations on arbitrarily-precise inputs and produce outputs, possibly after an initial delay, as rapidly as it reads its inputs. Such a representation system would thus limit hardware complexity, support use of variable precision, and facilitate parallel computation.

Chapter 8

Task Notes

This chapter ties up loose ends from Task 5 and our interim report [ORA88]. The chapter notes efforts planned in our interim report that we changed or were unable to carry out, and corrects two errors in our interim report.

8.1 Interval Probabilities

We noted in our interim report that it would sometimes be very useful if one could assign probabilities to the distribution of ideal results in the intervals produced by interval operations. It might be much more valuable to know that there is a 99.73% chance that a value is between 7.876 and 7.877, for example, than to know with certainty that this value is between 4.5 and 9.3. We did not find anything useful when we investigated this possibility.

As we noted in our interim report, if one starts with uniform distributions and computes the probability distribution of the results of interval operations accurately, the amount of information that must be stored for an interval can become arbitrarily large. Further, most of this information is useless. We thus looked into using approximate probability distributions that can be parameterized simply. The only well-known distribution besides the uniform distribution that is parameterized simply and applies to values that range over an interval is the beta distribution, and it does not describe the results of interval operations in any natural way.

8.2 “Briggsian” Algorithms

We were unable to perform the work we planned to do on quantitative results for interval versions of “Briggsian” algorithms, which perform the algebraic operations on some pocket calculators. We were unable to find literature on these algorithms. The name we were given for them might have been nonstandard, and significant information about them might be proprietary.

8.3 Theoretical Floating-Point Speed Limits

As we noted in Chapter 3, we were unable to find conclusive answers to questions about the theoretical speed limits of floating-point arithmetic carried out both consistently with the IEEE standard and otherwise, though what we did find suggests that adherence to the IEEE standard theoretically causes a significant loss in speed for division and taking square roots.

We were also unable to find the extent to which parts of individual floating-point operations could be done in parallel or how effectively many different floating-point operations could be done in parallel, with the exception of the information on pipelined floating-point arithmetic with highly redundant mantissas given in Chapter 3. We found some information on how these questions are addressed in Cray supercomputers [HX85], but were unable to study it or other literature on parallel computing thoroughly.

8.4 Alternative Representations

We ran out of time on Task 5 before we learned enough about some of the alternative representation systems considered in the literature, particularly these two, to make meaningful evaluations of them.

Iterated interval arithmetic is apparently a technique for computing and recomputing interval bounds on desired quantities until these bounds become short enough to achieve a desired precision. This arithmetic is thus a particular form of constructive-real arithmetic. We were unable to critically examine the time and space costs of operations using this technique.

Aberth [Abe88] describes one method of producing progressively shorter interval bounds with a version of range arithmetic in which the range is limited to one digit in the position of the mantissa's least significant digit and the mantissa loses digits if the range becomes larger. His system simply repeats all calculations to a higher precision — a larger number of mantissa digits — until the final results contain a desired number of accurate digits.

The ACRITH package from IBM [BRR85,KM83b] seems to use a more sophisticated algorithm for refining interval approximations. It apparently represents a real as an initial value plus a potentially infinite series of progressively smaller correction terms. It apparently produces interval bounds on results by truncating these series of correction terms, and computes appropriate correction terms for the results of operations by forming symbolic products of series and evaluating the initial terms of these series to obtain bounds consistent with the current level of precision. We gathered these impressions of the ACRITH package from Kahan's [KL85] critique of it, a critique that emphasized its sometimes excessive uses of time and space.

The variable-length p -adic representation by Horspool and Hehner [HH78] might be a significant extension of the finite p -adic representation described in Section 5.7, but we suspect that it also suffers from the defect of not providing a natural means for discarding information.

8.5 Experiments on Boehm's Package

As we noted in Chapter 6, we were unable to test the time and memory-use performances of Boehm's constructive-real package in finding the determinant of a large matrix. As we also noted in Chapter 6, however, we did identify means for doing so, and did obtain an expert opinion from Boehm on the issue the determinant experiment was intended to address. We also obtained the Russell matrix arithmetic package and one of the new versions of the Russell compiler that contains C-callable versions of the constructive-real arithmetic routines, so we can perform the determinant experiment in the future if asked to do so.

8.6 Aircraft Interception Example

We intended to examine code computing an interception path for aircraft [VS86] for situations where highly-parallel computation or intermediate results computed by a constructive-real package might be useful. We abandoned work on this example because we could not figure out the physics assumed by the algorithm, and because the Reals project was asked to look at a hostile booster trajectory estimation algorithm [App87] instead.

We found one potential situation where on-demand precision, which is related to constructive-real arithmetic, might be useful in the small fragment of this code that we examined. In this situation, the code computes directions by dividing vectors by their norms, even though these norms can be 0 or near 0. The resulting directions are actually not significant in this case, though, so we considered the situation a programming error rather than a potential application for constructive-real arithmetic.

8.7 Errata

Chapter 3 of our interim report contains two technical errors. First, the real number whose standard continued fraction is $[1, 1, 1, \dots]$ is not $\sqrt{2}$, but the golden ratio $(1 + \sqrt{5})/2$. The correct standard continued fraction for $\sqrt{2}$ is $[1, 2, 2, \dots]$.

Our interim report also makes the statement, “This property [the best rational approximation property] has as a consequence that finite initial portions of numbers’ continued fraction representations produce, on average, better approximations to the numbers per amount of information stored than do any other rational representations, including ordinary base- b notation.” This statement was based on a misunderstanding of the “best rational approximation” property of continued fractions. The results by Matula and Kornerup [KM85] on the gap sizes between consecutive LCF values, given in Subsection 5.2.2, can be construed as saying that the representation efficiency of the LCF encoding of continued fractions is asymptotically the same as that of the ordinary binary fixed-point representation.

Bibliography

- [Abe88] Oliver Aberth. *Precise Numerical Analysis*. Wm. C. Brown Publishers, Dubuque, Iowa, 1988.
- [Ale86] G. Alefeld. On the convergence of some interval-arithmetic modifications of Newton's Method. *SIAM J. Num. Anal.*, 21:363-372, 1986.
- * [App87] Applied Technology Associates. *Hostile Booster Trajectory Estimation*, November 1987. RADC-TR-87-202, distribution limited to DOD and DOD contractors only.
- [ASS85] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Massachusetts, 1985.
- [Atk68] Daniel E. Atkins. Higher-radix division using estimates of the divisor and partial remainders. *IEEE Transactions on Computers*, C-17(10):925-934, 1968.
- [Atr65] A. J. Atrubin. A one-dimensional real-time iterative multiplier. *IEEE Transactions on Electronic Computers*, EC-14:394-399, June 1965.
- [BB87] Jonathan M. Borwein and Peter B. Borwein. *Pi and the AGM*. John Wiley & Sons, New York, 1987.
- [BDD80] H. Boehm, A. Demers, and J. Donahue. An informal description of Russell. Technical Report 80-430, Computer Science Department, Cornell University, Ithaca, New York, 1980.

- [BMS89] Mark Bickford, Charlie Mills, and Edward A. Schneider. Clio: An applicative language-based verification system. Technical report, Odyssey Research Associates, Ithaca, New York, March 1989.
- [Boe87] Hans-Juergen Boehm. Constructive real interpretation of numerical programs. In *Proceedings, SIGPLAN '87 Symposium*. Association for Computing Machinery, 1987.
- [Boe89] Hans-Juergen Boehm. Personal correspondence, January 1989. Electronic-mail note of 1/20/89.
- [BPTP87] B. K. Bose, L. Pei, G. S. Taylor, and D. A. Patterson. Fast multiply and divide for a VLSI floating-point unit. In *Proc. 8th IEEE Symp. Comp. Arith.*, pages 87-94, 1987.
- [BRR85] J. H. Bleher, A. E. Roeder, and S. M. Rump. ACRITH: High-accuracy arithmetic, an advanced tool for numerical computation. In *Proc. 7th IEEE Symp. Comp. Arith.*, pages 318-321, 1985.
- [Cod79] W. J. Cody. Impact of the proposed IEEE floating point standard on numerical software. In *ACM/SIGNUM Newsletter, special issue on the Proposed IEEE Floating-Point Standard*, pages 29-30, October 1979.
- [Cod81] W. J. Cody. A proposed standard for binary floating-point arithmetic. *Computer*, 14(3):63-68, March 1981.
- [Coo81] Jerome T. Coonen. Underflow and the denormalized numbers. *Computer*, 14(3):75-87, March 1981.
- [Dad65] L. Dadda. Some schemes for parallel multipliers. *Alta Frequenza*, 34(5):349-356, May 1965.
- [Dem81] James Demmel. Effects of underflow on solving linear systems. In *Proc. 5th IEEE Symp. Comp. Arith.*, pages 113-119, 1981.
- [Dem87] James W. Demmel. On error analysis in arithmetic with varying relative precision. In *Proc. 8th IEEE Symp. Comp. Arith.*, pages 148-152, 1987.

- [Dig81] Digital Equipment Corporation, Maynard, Massachusetts. *VAX Architecture Handbook*, 1981.
- [Fan87] Jan Fandrianto. Algorithm for high speed shared radix 4 division and radix 4 square root. In *Proc. 8th IEEE Symp. Comp. Arith.*, pages 73-79, 1987.
- [Fel79] Stuart I. Feldman. The impact of the proposed standard for floating point arithmetic on languages and systems. In *ACM/SIGNUM Newsletter, special issue on the Proposed IEEE Floating-Point Standard*, pages 31-32, October 1979.
- [FH65] A. P. French and A. M. Hudson. Physics — a new introductory course. Technical report, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1965.
- [FM85] Warren E. Ferguson, Jr. and David W. Matula. Rationally biased arithmetic. In *Proc. 7th IEEE Symp. Comp. Arith.*, pages 194-202, 1985.
- [FW79] Bob Fraley and Steve Walther. Proposal to eliminate denormalized numbers. In *ACM/SIGNUM Newsletter, special issue on the Proposed IEEE Floating-Point Standard*, pages 22-23, October 1979.
- [GK84] R. T. Gregory and E. V. Krishnamurthy. *Methods and Applications of Error-Free Computation*. Springer-Verlag, New York, 1984.
- [Gos72] R. W. Gosper. Item 101 in HAKMEM. Technical Report AIM239, Massachusetts Institute of Technology, Cambridge, Massachusetts, February 1972.
- [Han75] E. Hansen. A generalized interval arithmetic. *Lecture Notes in Computer Science*, 29:7-18, 1975.
- [HC78] Kai Hwang and T. P. Chang. An interleaved rational/radix arithmetic system for high-precision computations. In *Proc. 4th IEEE Symp. Comp. Arith.*, pages 15-24, 1978.
- [HC87] Tackdon Han and David A. Carlson. Fast area-efficient VLSI adders. In *Proc. 8th IEEE Symp. Comp. Arith.*, pages 49-56, 1987.

- [HH78] R. Nigel Horspool and Eric C. R. Hehner. Exact arithmetic using a variable-length p -adic representation. In *Proc. 4th IEEE Symp. Comp. Arith.*, pages 10–14, 1978.
- [HL85] Albert E. Hurd and Peter A. Loeb. *An Introduction to Nonstandard Real Analysis*. Academic Press, Orlando, Florida, 1985.
- [Hou81] David Hough. Applications of the proposed IEEE 754 standard for floating-point arithmetic. *Computer*, 14(3):70–74, March 1981.
- [HW60] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*. Clarendon Press, Oxford, fourth edition, 1960.
- [HX85] Kai Hwang and Zhiwei Xu. Multiprocessors for evaluating compound arithmetic functions. In *Proc. 7th IEEE Symp. Comp. Arith.*, pages 266–275, 1985.
- [IEE85] IEEE Standards Board. IEEE standard for binary floating-point arithmetic. Technical Report ANSI/IEEE Std 754-1985, The Institute of Electrical and Electronics Engineers, New York, New York, 1985.
- [Jon84] Simon L. Peyton Jones. Arbitrary precision arithmetic using continued fractions. Technical Report INDRA Note 1530, University College, London, January 1984.
- [KL85] W. Kahan and E. LeBlanc. Anomalies in the IBM ACRITH package. In *Proc. 7th IEEE Symp. Comp. Arith.*, pages 322–331, 1985.
- [KM81] Peter Kornerup and David W. Matula. An integrated rational arithmetic unit. In *Proc. 5th IEEE Symp. Comp. Arith.*, pages 233–240, 1981.
- [KM83a] Peter Kornerup and David W. Matula. Finite precision rational arithmetic: An arithmetic unit. *IEEE Transactions on Computers*, C-32(4):378–388, 1983.
- [KM83b] Ulrich W. Kulisch and Willard L. Miranker, editors. *A New Approach to Scientific Computation*. Academic Press, New York, 1983.

- [KM85] Peter Kornerup and David W. Matula. Finite precision lexicographic continued fraction number systems. In *Proc. 7th IEEE Symp. Comp. Arith.*, pages 207–214, 1985.
- [KM87] Peter Kornerup and David W. Matula. A bit-serial arithmetic unit for rational arithmetic. In *Proc. 8th IEEE Symp. Comp. Arith.*, pages 204–211, 1987.
- [KM88] Peter Kornerup and David W. Matula. An on-line arithmetic unit for bit-pipelined rational arithmetic. *Journal of Parallel and Distributed Computing*, 5:310–330, 1988.
- [Knu81] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1981.
- [KP79] W. Kahan and J. Palmer. On a proposed floating-point standard. In *ACM/SIGNUM Newsletter, special issue on the Proposed IEEE Floating-Point Standard*, pages 13–21, October 1979.
- [Lov86] László Lovász. *An Algorithmic Theory of Numbers, Graphs and Convexity*. Society for Industrial and Applied Mathematics, Philadelphia, 1986.
- [Mat85] Y. Matijasevich. A posteriori version of interval analysis. In M. Arató, I. Káta, and L. Varga, editors, *Proc. Fourth Hung. Computer Sci. Conf.*, pages 339–349, 1985.
- [MI81] Shouichi Matsui and Masao Iri. An overflow/underflow-free floating point representation of numbers. *Journal of Information Processing*, 4(3):123–133, 1981.
- [MK80] D. W. Matula and P. Kornerup. Foundations of finite precision rational arithmetic. In *Computing, Suppl. 2*, pages 85–111. Springer-Verlag, New York, 1980.
- [MK83] David W. Matula and Peter Kornerup. An order preserving finite binary encoding of the rationals. In *Proc. 6th IEEE Symp. Comp. Arith.*, pages 201–209, 1983.

- [MK85] David W. Matula and Peter Kornerup. Finite precision rational arithmetic: Slash number systems. *IEEE Transactions on Computers*, C-34(1):3-17, 1985.
- [Mor73] R. Morris. Tapered floating point: A new floating-point representation. *IEEE Transactions on Computers*, C-20(6):1678-1679, 1973.
- [Olv87] F. W. J. Olver. A closed computer arithmetic. In *Proc. 8th IEEE Symp. Comp. Arith.*, pages 139-143, 1987.
- [ORA87] ORA Staff. A mathematical theory of asymptotic computation. Technical report, Odyssey Research Associates, Ithaca, New York, October 1987.
- [ORA88] ORA Staff. Reals interim report: Task 5. Technical report, Odyssey Research Associates, Ithaca, New York, April 1988.
- [Pix82] Carl Pixley. *Recommendations for OMNI Arithmetic*. Burroughs Corporation, Austin, Texas, March 1982. Inter-office Correspondence.
- [PS79] Mary Payne and William Strecker. Draft proposal for a binary normalized floating point standard. In *ACM/SIGNUM Newsletter, special issue on the Proposed IEEE Floating-Point Standard*, pages 24-28, October 1979.
- [PSG87] Victor Peng, Sridhar Samudrala, and Moshe Gavrielov. On the implementation of shifters, multipliers and dividers in VLSI floating point units. In *Proc. 8th IEEE Symp. Comp. Arith.*, pages 95-102, 1987.
- [Reg88] Ken Regan. Minimum-complexity pairing functions. Technical Report Mathematical Sciences Institute TR #88-97, Cornell University, Ithaca, New York, September 1988.
- [Rog67] Hartley Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. Mc-Graw Hill, New York, 1967.

- [SB80] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer-Verlag, New York, 1980.
- [Sha87] Ramautar Sharma. Area-time efficient arithmetic elements for VLSI systems. In *Proc. 8th IEEE Symp. Comp. Arith.*, pages 57-62, 1987.
- [TE77] Kishor S. Trivedi and Miloš D. Ercegovac. On-line algorithms for division and multiplication. *IEEE Transactions on Computers*, C-26(7):681-687, 1977.
- [TP81] George S. Taylor and David A. Patterson. VAX hardware for the proposed IEEE floating-point standard. In *Proc. 5th IEEE Symp. Comp. Arith.*, pages 190-196, 1981.
- [Tur76] D. A. Turner. The SASL language manual. Technical report, University of St. Andrews, December 1976.
- [VS86] Hendrikus G. Visser and Josef Shinar. A highly accurate feedback approximation for horizontal variable-speed interceptions. *J. Guidance*, 9(6):691-698, 1986.
- [Wal64] C. S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, EC-13:14-17, February 1964.
- [WE81] O. Watanuki and M. D. Ercegovac. Floating-point on-line arithmetic: Algorithms. In *Proc. 5th IEEE Symp. Comp. Arith.*, pages 81-86, 1981.
- [Yos83] Kaoru Yoshida. Floating-point recurring rational arithmetic system. In *Proc. 6th IEEE Symp. Comp. Arith.*, pages 194-200, 1983.

* RADC-TR-90-53, Vol I, Formal Verification of Mathematical Software, distribution limited to U.S. Govt Agencies and their contractors. May 1990.

Appendix A

IEEE Interval Arithmetic

This appendix describes a particular version of interval arithmetic that can be implemented on any machine meeting the IEEE standard for binary floating-point arithmetic [IEE85]. Appendix B gives an implementation of this arithmetic for Sun computers. The “arithmetic” includes operations such as interval intersection that do not correspond to operations on real numbers but are often used in interval algorithms, as in the Interval Newton’s Method algorithm implemented by code in Appendix E.

A.1 Extended Real-Number Arithmetic

Define the set \mathbf{R}' by $\mathbf{R}' = (\mathbf{R} \setminus \{0\}) \cup \{+0, -0, +\infty, -\infty\}$, where $+0$, -0 , $+\infty$ and $-\infty$ are new symbols. The values $+0$ and -0 behave as positive and negative infinitesimals, respectively, and the values $+\infty$ and $-\infty$ behave as positive and negative infinity. These new values correspond to possible values in IEEE floating-point arithmetic, which is described in this appendix’s Section 3. Extend the usual order $<$ on \mathbf{R} to a similar order $<'$ on \mathbf{R}' by saying that for every positive real x , $-\infty <' -x <' -0 <' +0 <' x <' +\infty$.

Define the set \mathbf{R}'' by $\mathbf{R}'' = \mathbf{R}' \cup \{\text{NaN}, +\text{NaN}, -\text{NaN}\}$. The NaN value corresponds to the “not a number” value used in IEEE floating-point arithmetic as the result of invalid, completely indeterminate operations. The $+\text{NaN}$ and $-\text{NaN}$ values correspond to values that are known only to be pos-

itive or negative, respectively. They do not correspond to values in IEEE arithmetic, but will be used in this appendix's Section 4 to define operations on intervals.

Extend the operations in $\{+, -, \cdot, /\}$ on \mathbf{R} to \mathbf{R}'' consistently with the theory of limits. In particular,

$$\begin{aligned}-(\pm\infty) &= \mp\infty \\ 1/(\pm 0) &= \pm\infty \\ 1/(\pm\infty) &= \pm 0 \\ (\pm\infty) + (\mp\infty) &= \text{NaN} \\ (\pm\infty) - (\pm\infty) &= \text{NaN} \\ +0 \cdot +\infty &= +\text{NaN} \\ +0 \cdot -\infty &= -\text{NaN} \\ -0 \cdot +\infty &= -\text{NaN} \\ -0 \cdot -\infty &= +\text{NaN} \\ \pm 0 / +0 &= \pm\text{NaN} \\ \pm 0 / -0 &= \mp\text{NaN} \\ \pm\infty / +\infty &= \pm\text{NaN} \\ \pm\infty / -\infty &= \mp\text{NaN}\end{aligned}$$

Let any operation with NaN as one of its arguments have NaN as its result. Let an operation with +NaN or -NaN as one of its arguments have the result consistent with only that argument's sign being known. In particular, $-(\pm\text{NaN}) = \mp\text{NaN}$, $+\text{NaN} + +\text{NaN} = +\text{NaN}$, and $+\text{NaN} + -\text{NaN} = \text{NaN}$.

Do not extend the order $<'$ to \mathbf{R}'' . In IEEE floating-point arithmetic, any comparison operation involving NaN returns NaN as its result; the comparison operations do not even identify NaN as being equal to itself.

A.2 IEEE Floating-Point Arithmetic

Following the IEEE standard, let $M_{m,n}$ for positive integers m and n be the set of all real numbers of the form

$$\pm \text{significand} \cdot 2^{\text{exponent}},$$

where $0 < \text{significand} < 2$, *significand* is an integral multiple of 2^{-n} , and $-m \leq \text{exponent} \leq m$. The values of m and n vary on different machines.

Assume that m and n are fixed, and let $M = M_{m,n} \cup \{+0, -0, +\infty, -\infty\}$. M denotes the machine-representable "numerical" values on a machine supporting IEEE-standard floating-point arithmetic.

Let a *rounding* be a function $\square : \mathbf{R}' \rightarrow M$ satisfying:

$$\begin{aligned} \forall x \in M \quad (\square x = x) \quad \text{and} \\ \forall x, y \in \mathbf{R}' \quad (x \leq y \Rightarrow \square x \leq \square y). \end{aligned}$$

Note that these properties have as a consequence that $\square x = x$ or $\square x$ is one of the two representable values closest to x .

Say that a rounding \square is *upward* if $\square x \geq x$, and *downward* if $\square x \leq x$. The downward rounding of zero is -0 and the upward rounding of zero is $+0$. The IEEE standard also defines two other types of roundings, *toward zero* and *to-nearest*, and calls for the maintenance of a *rounding mode* that selects one of these four roundings as the current rounding. The programmer can change the rounding mode at will. For $\star \in \{+, -, \cdot, /\}$, the standard calls for the corresponding machine operation \star_M to satisfy

$$x \star_M y = \square(x \star y)$$

for all machine-representable reals x and y , where \square is the current rounding. This note will not require the full generality of the IEEE standard, but will use upward and downward roundings in defining the interval arithmetic computations. The asymptotic version of the interval semantics also depends on properties of these roundings.

A.3 Machine Interval Arithmetic

For values a_1 and a_2 in \mathbf{R}' , with $a_1 \leq' a_2$, call a subset of \mathbf{R}' of the form

$$A = [a_1, a_2] = \{t \in \mathbf{R}' \mid a_1 \leq t \leq a_2\}$$

an *interval*. Note that $+0$ and -0 are distinct as possible endpoints. Let the special value EMPTY, denoting the empty set, be an interval, and let POSINF and NEGINF denote the intervals $[+\infty, +\infty]$ and $[-\infty, -\infty]$, respectively. POSINF can be thought of as an infinite interval of positive

reals whose magnitudes are all too large to be machine-representable, and NEGINF as a similar interval of negative numbers.

The set of possible intervals contains intervals that exactly correspond to each of the possible values representable in IEEE arithmetic, as well as to the +NaN and -NaN values introduced in this appendix's Section 2. The real number x corresponds to the point-interval $[x, x]$; the values $+\infty$ and $-\infty$ correspond to the intervals POSINF and NEGINF, respectively; the values +NaN and -NaN correspond to the intervals $[+0, +\infty]$ and $[-\infty, -0]$, respectively; and NaN corresponds to the interval $[-\infty, +\infty]$.

For $\star \in \{+, -, \cdot, /\}$ and intervals A and B , define the interval operation $A \star B$ as follows: Consider the intermediate set VALUES defined by

$$\text{VALUES} = \{a \star b \mid a \in A, b \in B\},$$

where the operations are performed in \mathbf{R}'' . Define new intermediate sets KNOWN and UNKNOWN by

$$\text{KNOWN} = \text{VALUES} \setminus \{+\text{NaN}, -\text{NaN}, \text{NaN}\},$$

and

$$\text{UNKNOWN} = \text{VALUES} \cap \{+\text{NaN}, -\text{NaN}, \text{NaN}\}.$$

Replace the values +NaN, -NaN and NaN in UNKNOWN by the intervals $[+0, +\infty]$, $[-\infty, -0]$ and $[-\infty, +\infty]$, respectively, and let UNCERTAIN be the union of the members of the resulting set. Finally, let $A \star B$ be the union of KNOWN and UNCERTAIN.

Now define the *machine-representable* intervals and the operations on them. Let \mathbf{M} be the set of machine-representable "numerical" values for a particular machine. Let \uparrow (\downarrow) be an upward (downward) rounding that rounds to values in \mathbf{M} . Define a *conservative rounding* on intervals by

$$\uparrow A = \uparrow [a_1, a_2] = [\downarrow a_1, \uparrow a_2].$$

Finally, define the *machine interval-arithmetic operations* for $\star \in \{+, -, \cdot, /\}$ by

$$A \star_M B = \uparrow (A \star B).$$

The rest of this appendix will only consider machine-representable intervals, so take "interval" as an abbreviation for "machine-representable interval" from now on. Note that all operations are defined for all possible pairs

of intervals. The ability to compute interval operations is dependent on the availability of appropriate upward and downward rounding functions; such functions are available in hardware for machines meeting the IEEE floating-point standard.

A.4 Semantics of Interval Operations

Assume that the machine on which the interval operations are performed maintains a finite collection of flags called the *status*, where each flag in the status has value SET or CLEAR. Following the IEEE standard, this appendix considers *inexact*, *invalid-operation*, *divide-by-zero*, *overflow* and *underflow* as possible status flags. Assume programs can test and set the state of the status flags by making appropriate system calls, and assume, following the IEEE standard, that all status flags are initially CLEAR but that once a status flag becomes set it stays set until the program makes a system call to reset it to CLEAR. Say that an operation *causes an exception* if it causes the status flag for a condition to be set if that flag is not already set. Let each status flag name the exception consisting of causing that flag to become SET. Note that a single operation can cause more than one exception.

Further, assume that the constants $+0$, -0 , 1 , *EMPTY*, *POSINF* and *NEGINF*, denoting the intervals $[+0, +0]$, $[-0, -0]$, $[1, 1]$, $[1, 0]$, $[+\infty, +\infty]$ and $[-\infty, -\infty]$, respectively, are available to programs. (The subroutine *doinits* in Appendix B creates the necessary constants without causing any exceptions in the process; these constants will presumably be more readily available after extensions to C are chosen to take advantage of items available in IEEE arithmetic.) Also assume the binary functions $+$, $-$, \cdot , $/$, \cup and \cap , and the unary functions *length*, *left-end*, *right-end* and *mid-point* are available.

For intervals Q and R , the operation $Q \star R$, for $\star \in \{+, \cdot, -, /\}$, is as defined above. The operation causes the following exceptions in the given conditions; otherwise the transition does not cause any exceptions:

1. An *inexact* exception occurs whenever the machine interval produced as the result of the operation is not the exact result for the operation.

2. An overflow or underflow exception can occur whenever an *inexact* exception occurs. Whether or not one of these exceptions occurs on a particular operation with particular arguments will vary with the machine, the operation, the arguments, and the details of exactly how the interval operations are implemented.
3. A divide-by-zero exception occurs whenever the interval being divided by contains either $+0$ or -0 .
4. An invalid-operation exception occurs in each of these cases: one or both of the arguments to an operation is *EMPTY*; the operation is the sum of two intervals and one has $+\infty$ as its upper bound and the other has $-\infty$ as its lower bound; the operation is the difference of two intervals and both intervals have $+\infty$ ($-\infty$) as the same bound (i.e., both upper or both lower); the operation is interval multiplication, one of the intervals is not finite (i.e., if it has $+\infty$ or $-\infty$ as a bound), and the other contains $+0$ or -0 ; the operation is interval division, both intervals contain some zero (i.e., $+0$ or -0), or both have at least one infinite bound.

The \cup of two intervals is the interval from the least point to the greatest point (under the order $<'$) in the union of the intervals. The \cap of two intervals is the (possibly *EMPTY*) intersection of the intervals. Neither operation causes any exceptions.

The length of an interval is an interval containing the interval's length. If the interval is of infinite length, which it will be if it is *POSINF* or *NEGINF*, then its length is *POSINF*. The length of *EMPTY* is taken to be 0. If the value of a length operation is not a point interval or *POSINF* then the operation causes an *inexact* exception; otherwise it causes no exceptions.

The left-end (right-end) of a nonempty interval is the point interval containing the possibly-infinite least (greatest) point in the interval. The left and right ends of *EMPTY* are taken to be *POSINF* and *NEGINF*, respectively, the left and right ends of *POSINF* are both taken to be *POSINF*, and the left and right ends of *NEGINF* are both taken to be *NEGINF*. The left-end operation on an *EMPTY* or *POSINF* interval, and the right-end operation on an *EMPTY* or *NEGINF* interval, cause an invalid-operation exception; otherwise these operations cause no exceptions.

Appendix B

IEEE Interval Operations

This appendix contains a copy of the file included as `intops.c` in the interval algorithm programs given in Appendices C and E. It implements the interval operations specified in Appendix A. It also contains the function `litprint` for showing the binary values of IEEE double-precision floating-point values as they are implemented on Sun machines, and the function `doinits` for creating infinite constants without causing exceptions.

This code is written to run under Release 3.5 of the Sun UNIX 4.2 operating system. It makes the changes in the rounding mode necessary to give optimally-rounded intervals with calls to the `fpmode_` system call, which has been replaced with a different system call in more recent releases of the Sun operating system.

The code begins on the next page.

```
/* LITMAX should be as large as the largest number of */  
/* calls to litprint in a single invocation of printf */
```

```
#define LITMAX 10  
#define LITLNGTH 20
```

```
char *litprint(x)  
double x;  
{  
static char litstrings[LITMAX][LITLNGTH];  
static int litindex = 0; /* compiler init needed */  
char *pstring,*sprintf();  
unsigned short *px;  
  
px = (unsigned short *) &x;  
pstring = litstrings[litindex];  
  
(void) sprintf(pstring,"%04x %04x %04x %04x",  
               *px,*px+1,*px+2,*px+3);  
  
++litindex;  
if(litindex == LITMAX)  
    litindex = 0;  
  
return(pstring);  
}
```

```
void doinits()  
{  
static unsigned INFPART = 0x7ff00000;  
unsigned *punsign;  
double temp;  
  
punsign = (unsigned *) &temp;  
*punsign = INFPART;  
*(punsign+1) = 0;
```

```

    PLUINF = temp;
    MININF = -temp;
    POSINF.l = PLUINF;
    POSINF.r = PLUINF;
    NEGINF.l = MININF;
    NEGINF.r = MININF;
}

```

```

struct interval intsum(int1,int2)
struct interval int1,int2;
{
    struct interval result;

    if(int1.l > int1.r || int2.l > int2.r) { /* EMPTY argument */
        result.l = PLUINF + MININF;          /* Cause exception */
        result = EMPTY;                      /* EMPTY result */
    }
    else {
        newmode = 2*64 + 2*16;
        oldmode = fpmode_(&newmode);
        result.l = int1.l + int2.l;
        newmode = 2*64 + 3*16;
        newmode = fpmode_(&newmode);
        result.r = int1.r + int2.r;
        newmode = fpmode_(&oldmode);
    }
    return(result);
}

```

```

struct interval intdiff(int1,int2)
struct interval int1,int2;
{
    struct interval result;

    if(int1.l > int1.r || int2.l > int2.r) { /* EMPTY argument */

```



```

        result.l = PLUINF + MININF;          /* Cause exception */
        result = EMPTY;                     /* EMPTY result */
    }
else {
    newmode = 2*64 + 2*16;
    oldmode = fpmode_(&newmode);
    result.l = int1.l - int2.r;
    newmode = 2*64 + 3*16;
    newmode = fpmode_(&newmode);
    result.r = int1.r - int2.l;
    newmode = fpmode_(&oldmode);
}
return(result);
}

```

```

struct interval intprod(int1,int2)
struct interval int1,int2;
{
    int i,j,isinf();
    double temp1,temp2,temp3,temp4,low,high;
    struct interval result;
    unsigned sign1,sign2,*punsign;

    if(int1.l > int1.r || int2.l > int2.r) { /* EMPTY argument */
        result.l = PLUINF + MININF;          /* Cause exception */
        result = EMPTY;                     /* EMPTY result */
    }
else {
    if(int1.l==0.0||
       int1.r==0.0||
       int2.l==0.0||
       int2.r==0.0||
       isinf(int1.l)||
       isinf(int1.r)||
       isinf(int2.l)||
       isinf(int2.r)

```

```

) {

/* "Special calculations needed" case */

low = PLUINF;
high = MININF;
for(i=0; i<2; ++i) {
    temp1 = (i==0)?int1.l:int1.r;
    for(j=0; j<2; ++j) {
        temp2 = (j==0)?int2.l:int2.r;
        if((temp1 == 0.0 && isinf(temp2)) ||
            (temp2 == 0.0 && isinf(temp1))) {

            /* NaN case */

            punsign = (unsigned *) &temp1;
            sign1 = *punsign >> 31;
            punsign = (unsigned *) &temp2;
            sign2 = *punsign >> 31;
            if((sign1 && sign2) ||
                (!sign1 && !sign2)) {
                if(low > 0.0)
                    low = 0.0;
                high = PLUINF;
            }
            else {
                if(high < 0.0) {
                    high = 0.0;
                    high *= -1;
                }
                low = MININF;
            }
        }
    }
}
else {

```

```

/* Normal case */

newmode = 2*64 + 2*16;
oldmode = fpmode_(&newmode);
temp3 = temp1 * temp2;
newmode = 2*64 + 3*16;
newmode = fpmode_(&newmode);
temp4 = temp1 * temp2;
if(temp3 < low)
    low = temp3;
else {
    if(temp3==0.0 && low==0.0) {
        punsign = (unsigned *) &temp3;
        sign1 = *punsign >> 31;
        punsign = (unsigned *) &low;
        sign2 = *punsign >> 31;
        if(sign1 && !sign2)
            low = temp3;
    }
}
if(temp4 > high)
    high = temp4;
else {
    if(temp4==0.0 && high==0.0) {
        punsign = (unsigned *) &temp4;
        sign1 = *punsign >> 31;
        punsign = (unsigned *) &high;
        sign2 = *punsign >> 31;
        if(!sign1 && sign2)
            high = temp4;
    }
}
}
}
}
result.l = low;
result.r = high;

```

```

        /* cause exception if called for */

        if((int1.l <= 0.0 &&
int1.r >= 0.0 &&
        (isinf(int2.l) || isinf(int2.r))
        ) ||
        (int2.l <= 0.0 &&
        int2.r >= 0.0 &&
        (isinf(int1.l) || isinf(int1.r))
        )
        )
            temp3 = PLUINF + MININF;

    }
else {

    /* "Old-fashioned code sufficient" case */

    newmode = 2*64 + 2*16;
    oldmode = fpmode_(&newmode);
    if(int1.l > 0.0) {
        if(int2.l > 0.0) {
            result.l = int1.l * int2.l;
            newmode = 2*64 + 3*16;
            newmode = fpmode_(&newmode);
            result.r = int1.r * int2.r;
        }
        else {
            if(int2.r < 0.0) {
                result.l = int1.r * int2.l;
                newmode = 2*64 + 3*16;
                newmode = fpmode_(&newmode);
                result.r = int1.l * int2.r;
            }
            else {
                result.l = int1.r * int2.l;

```

```

        newmode = 2*64 + 3*16;
        newmode = fpmode_(&newmode);
        result.r = int1.r * int2.r;
    }
}
else {
    if(int1.r < 0.0) {
        if(int2.l > 0.0) {
            result.l = int1.l * int2.r;
            newmode = 2*64 + 3*16;
            newmode = fpmode_(&newmode);
            result.r = int1.r * int2.l;
        }
        else {
            if(int2.r < 0.0) {
                result.l = int1.r * int2.r;
                newmode = 2*64 + 3*16;
                newmode = fpmode_(&newmode);
                result.r = int1.l * int2.l;
            }
            else {
                result.l = int1.l * int2.r;
                newmode = 2*64 + 3*16;
                newmode = fpmode_(&newmode);
                result.r = int1.l * int2.l;
            }
        }
    }
    else {
        if(int2.l > 0.0) {
            result.l = int1.l * int2.r;
            newmode = 2*64 + 3*16;
            newmode = fpmode_(&newmode);
            result.r = int1.r * int2.r;
        }
        else {

```

```

        if(int2.r < 0.0) {
            result.l = int1.r * int2.l;
            newmode = 2*64 + 3*16;
            newmode = fpmode_(&newmode);
            result.r = int1.l * int2.l;
        }
        else {
            temp1 = int1.l * int2.r;
            temp2 = int1.r * int2.l;
            result.l = (temp1 <= temp2)?temp1:temp2;
            newmode = 2*64 + 3*16;
            newmode = fpmode_(&newmode);
            temp1 = int1.l * int2.l;
            temp2 = int1.r * int2.r;
            result.r = (temp1 >= temp2)?temp1:temp2;
        }
    }
}
newmode = fpmode_(&oldmode);
}
}
return(result);
}

struct interval intquot(int1,int2)
struct interval int1,int2;
{
    int i,j,isinf();
    double temp1,temp2,temp3,temp4,low,high;
    struct interval result;
    unsigned sign1,sign2,*punsign;

    if(int1.l > int1.r || int2.l > int2.r) { /* EMPTY argument */
        result.l = PLUINF + MININF;          /* Cause exception */
        result = EMPTY;                      /* EMPTY result */
    }

```

```

    }
else {
    if(int1.l==0.0||
       int1.r==0.0||
       int2.l==0.0||
       int2.r==0.0||
       isinf(int1.l)||
       isinf(int1.r)||
       isinf(int2.l)||
       isinf(int2.r)||
       (int2.l < 0.0 && int2.r > 0.0)
    ) {

        /* "Special calculations needed" case */

        if(int2.l < 0.0 && int2.r > 0.0) {
            result.l = MININF;
            result.r = PLUINF;
            temp3 = 0.0;
            temp4 = 1.0/temp3;    /* give 0-divide exception */
        }
        else {
            low = PLUINF;
            high = MININF;
            for(i=0; i<2; ++i) {
                temp1 = (i==0)?int1.l:int1.r;
                for(j=0; j<2; ++j) {
                    temp2 = (j==0)?int2.l:int2.r;
                    if((temp1 == 0.0 && temp2 == 0.0) ||
                       (isinf(temp1) && isinf(temp2)))
                    ) {

                        /* NaN case */

                        punsign = (unsigned *) &temp1;
                        sign1 = *punsign >> 31;
                        punsign = (unsigned *) &temp2;

```

```

sign2 = *punsign >> 31;
if((sign1 && sign2)||
    (!sign1 && !sign2)
    ) {
    if(low > 0.0)
        low = 0.0;
    high = PLUINF;
}
else {
    if(high < 0.0) {
        high = 0.0;
        high *= -1;
    }
    low = MININF;
}

/* cause 0-divide if needed */

if(temp2 == 0.0) {
    temp3 = 0.0;
    temp4 = 1.0/temp3;
}

}
else {

    /* Normal case */

    newmode = 2*64 + 2*16;
    oldmode = fpmode_(&newmode);
    temp3 = temp1 / temp2;
    newmode = 2*64 + 3*16;
    newmode = fpmode_(&newmode);
    temp4 = temp1 / temp2;
    if(temp3 < low)
        low = temp3;
    else {

```



```

        if(temp3==0.0&&low==0.0) {
            punsign = (unsigned *) &temp3;
            sign1 = *punsign >> 31;
            punsign = (unsigned *) &low;
            sign2 = *punsign >> 31;
            if(sign1 && !sign2)
                low = temp3;
        }
    }
    if(temp4 > high)
        high = temp4;
    else {
        if(temp4==0.0&&high==0.0) {
            punsign = (unsigned *) &temp4;
            sign1 = *punsign >> 31;
            punsign = (unsigned *) &high;
            sign2 = *punsign >> 31;
            if(!sign1 && sign2)
                high = temp4;
        }
    }
}

result.l = low;
result.r = high;
}

/* cause invalid-op exception if needed */

if((int1.l <= 0.0 &&
    int1.r >= 0.0 &&
    int2.l <= 0.0 &&
    int2.r >= 0.0
) ||
((isinf(int1.l) || isinf(int1.r)) &&
(isinf(int2.l) || isinf(int2.r))

```

```

    )
    )
    temp3 = PLUINF + MININF;
}
else {

    /* "Old-fashioned code sufficient" case */

    newmode = 2*64 + 2*16;
    oldmode = fpmode_(&newmode);
    if(int1.l > 0.0) {
        if(int2.l > 0.0) {
            result.l = int1.l / int2.r;
            newmode = 2*64 + 3*16;
            newmode = fpmode_(&newmode);
            result.r = int1.r / int2.l;
        }
        else {
            result.l = int1.r / int2.r;
            newmode = 2*64 + 3*16;
            newmode = fpmode_(&newmode);
            result.r = int1.l / int2.l;
        }
    }
    else {
        if(int1.r < 0.0) {
            if(int2.l > 0.0) {
                result.l = int1.l / int2.l;
                newmode = 2*64 + 3*16;
                newmode = fpmode_(&newmode);
                result.r = int1.r / int2.r;
            }
            else {
                result.l = int1.r / int2.l;
                newmode = 2*64 + 3*16;
                newmode = fpmode_(&newmode);
            }
        }
    }
}

```

```

        result.r = int1.l / int2.r;
    }
}
else {
    if(int2.l > 0.0) {
        result.l = int1.l / int2.l;
        newmode = 2*64 + 3*16;
        newmode = fpmode_(&newmode);
        result.r = int1.r / int2.l;
    }
    else {
        result.l = int1.r / int2.r;
        newmode = 2*64 + 3*16;
        newmode = fpmode_(&newmode);
        result.r = int1.l / int2.r;
    }
}
}
newmode = fpmode_(&oldmode);
}
}
return(result);
}

struct interval intsqrt(intx)
struct interval intx;
{
    struct interval result;

    if(intx.l > intx.r || intx.r < 0.0) {
        intx.l = PLUINF + MININF;    /* generate exception */
        result = EMPTY;
    }
    else {
        newmode = 2*64 + 2*16;        /* round lower end down */
        oldmode = fpmode_(&newmode);
    }
}

```

```

    if(intx.l < 0.0) {
        result.l = 0.0;
        result.l *= -1.0;
    }
    else
        result.l = sqrt(intx.l);
    newmode = 2*64 + 3*16; /* round upper end up */
    newmode = fpmode_(&newmode);
    result.r = sqrt(intx.r);
    newmode = fpmode_(&oldmode);
}
return(result);
}

```

```

struct interval intunion(x,y)
struct interval x,y;
{
    struct interval result;

    if(x.l > x.r)
        result = y;
    else {
        if(y.l > y.r)
            result = x;
        else {
            result.l = (x.l <= y.l)?x.l:y.l;
            result.r = (x.r >= y.r)?x.r:y.r;
        }
    }
    return(result);
}

```

```

struct interval intinter(x,y)
struct interval x,y;
{

```

```

struct interval result;

result.l = (x.l >= y.l)?x.l:y.l;
result.r = (x.r <= y.r)?x.r:y.r;
if(result.l > result.r)
    result = EMPTY;
return(result);
}

```

```

struct interval intlength(intx)
struct interval intx;
{
    struct interval result;

    if(intx.l >= intx.r) {
        result.l = 0.0;
        result.r = 0.0;
    }
    else {
        newmode = 2*64 + 2*16;
        oldmode = fpmode_(&newmode);
        result.l = intx.r - intx.l;
        newmode = 2*64 + 3*16;
        newmode = fpmode_(&newmode);
        result.r = intx.r - intx.l;
        newmode = fpmode_(&oldmode);
    }
    return(result);
}

```

```

struct interval leftend(intx)
struct interval intx;
{
    struct interval result;

```

```

if(intx.l > intx.r) {
    intx.l = PLUINF + MININF;    /* generate exception */
    result = POSINF;
}
else {
    if(intx.l == PLUINF)
        intx.l = PLUINF + MININF; /* generate exception */
    result.l = intx.l;
    result.r = intx.l;
}
return(result);
}

```

```

struct interval rightend(intx)
struct interval intx;
{
    struct interval result;

    if(intx.l > intx.r) {
        intx.l = PLUINF + MININF;    /* generate exception */
        result = NEGINF;
    }
    else {
        if(intx.r == MININF)
            intx.l = PLUINF + MININF; /* generate exception */
        result.l = intx.r;
        result.r = intx.r;
    }
    return(result);
}

```

Appendix C

FFT Multiplication Programs

The following C programs implement versions of an algorithm to use Fast Fourier Transforms to multiply large integers. The first two give interval versions of this algorithm, and the last one gives a scalar version. One interval program is for a Sun and the other is for a VAX. The scalar program is for a Sun but can be easily modified, by following comments in the code, to behave similarly on a VAX. If these programs are run with reasonably small integers as inputs, the exact results of the Fast Fourier Transform algorithm can be calculated with machine integer multiplication, so the lengths of the intervals the programs produce are greater than zero, or the floating-point answers they produce differ from the exact results, only because of error accumulated in doing the calculations.

As explained in Chapter 2, these programs and their outputs show that, at least for moderately-complicated calculations, having additional bits in the mantissas of the floating-point values being calculated has a greater influence on precision than having optimally-rounded results does. They also illustrate that interval algorithms obtained by simply reinterpreting the values of variables as intervals and the arithmetic operations on these variables as corresponding interval operations tend to produce results whose error bounds are much larger than the actual errors in the corresponding floating-point calculations.

C.1 IEEE Interval FFT Multiply

The first program uses IEEE double-precision floating-point arithmetic as implemented on a Sun 3/60 having an MC68881 coprocessor with mask A93N. Its interval operations force results to be rounded correctly by calling `fpmode_` under Release 3.5 of the Sun UNIX 4.2 operating system. Its interval operations, obtained by including the file `intops.c`, are given in Appendix B; their semantics is specified in Appendix A.

```
#include <stdio.h>
#include <math.h>

unsigned oldmode,newmode,fpmode_();

double PLUINF,MININF;

struct interval {double l,r;};
struct interval EMPTY = {1.0, 0.0};
struct interval ZERO = {0.0, 0.0};
struct interval ONE = {1.0, 1.0};
struct interval POSINF,NEGINF;

#define KKNUTH 8
#define LKNUTH 8
#define BIGKKNUTH 256

int reverse[BIGKKNUTH];

struct interval TWO = {2.0,2.0};
struct interval INT256 = {256.0,256.0};

struct complex {struct interval x,y;} wpow[BIGKKNUTH],
               w2pow[KKNUTH+1];

main()
{
```



```

void doinits();
char *litprint();
unsigned char *pd;
unsigned num0,num1,product;
int i,j,k,rk;
double power2;
struct interval temp;
struct interval intsum(),intdiff(),
intprod(),intquot(),intsqrt();
struct complex s,t,f1[BIGKKNUTH],f2[BIGKKNUTH];

doinits(); /* form infinite constants causing no exceptions */
for(i=0; i < BIGKKNUTH; ++i) {
    k = i;
    rk = 0;
    for(j=1; j < KKNUTH; ++j) {
        if(k&1)
            rk += 1;
        k >>= 1;
        rk <<= 1;
    }
    if(k&1)
        rk += 1;
    reverse[i] = rk;
}

w2pow[1].x.l = -1.0;
w2pow[1].x.r = -1.0;
w2pow[1].y.l = 0.0;
w2pow[1].y.r = 0.0;
w2pow[2].x.l = 0.0;
w2pow[2].x.r = 0.0;
w2pow[2].y.l = 1.0;
w2pow[2].y.r = 1.0;
for(i=3; i <= KKNUTH; ++i) {
    w2pow[i].x = intsqrt(
        intquot(

```

```

                                intsum(ONE,w2pow[i-1].x),
                                TWO
                                )
                                );
w2pow[i].y = intsqrt(
                                intquot(
                                    intdiff(ONE,w2pow[i-1].x),
                                    TWO
                                )
                                );
    }
w2pow[0] = w2pow[KKNUTH];

for(i=0; i < BIGKKNUTH; ++i) {
    s.x = ONE;
    s.y = ZERO;
    j = i;
    k = 0;
    while(j != 0) {
        if(j&1) {
            t.x = intdiff(
                                intprod(s.x,w2pow[KKNUTH-k].x),
                                intprod(s.y,w2pow[KKNUTH-k].y)
                                );
            t.y = intsum(
                                intprod(s.x,w2pow[KKNUTH-k].y),
                                intprod(s.y,w2pow[KKNUTH-k].x)
                                );
            s = t;
        }
        j >>= 1;
        ++k;
    }
    wpow[i] = s;
}

(void) fprintf(stderr,"input first positive number: ");

```

```

(void) scanf("%d",&num0);
(void) fprintf(stderr,"input second positive number: ");
(void) scanf("%d",&num1);

power2 = (double) (1<<(KKNUTH + LKNUTH)); /* exact */
for(i=0,pd = ((unsigned char *) &num0) + 3;
    i < 4;
    ++i,--pd) {
    f1[i].x.l = ((double) *pd)/power2; /* exact */
    f1[i].x.r = f1[i].x.l;
    f1[i].y.l = 0.0;
    f1[i].y.r = 0.0;
}
for(i=4; i < BIGKKNUTH; ++i) {
    f1[i].x.l = 0.0;
    f1[i].x.r = 0.0;
    f1[i].y.l = 0.0;
    f1[i].y.r = 0.0;
}
for(i=0,pd = ((unsigned char *) &num1) + 3;
    i < 4;
    ++i,--pd) {
    f2[i].x.l = ((double) *pd)/power2; /* exact */
    f2[i].x.r = f2[i].x.l;
    f2[i].y.l = 0.0;
    f2[i].y.r = 0.0;
}
for(i=4; i < BIGKKNUTH; ++i) {
    f2[i].x.l = 0.0;
    f2[i].x.r = 0.0;
    f2[i].y.l = 0.0;
    f2[i].y.r = 0.0;
}

fft(f1);
fft(f2);

```

```

for(i=0; i < BIGKKNUTH; ++i) {
    temp = f1[i].x;
    f1[i].x = intdiff(
        intprod(temp,f2[i].x),
        intprod(f1[i].y,f2[i].y)
    );
    f1[i].y = intsum(
        intprod(temp,f2[i].y),
        intprod(f1[i].y,f2[i].x)
    );
}

fft(f1);

f2[0].x.l = (f1[0].x.l)/((double) BIGKKNUTH); /* exact */
f2[0].x.r = (f1[0].x.r)/((double) BIGKKNUTH); /* exact */
for(i=1; i < BIGKKNUTH; ++i) {
    f2[i].x.l = (f1[BIGKKNUTH-i].x.l)/((double) BIGKKNUTH);
    f2[i].x.r = (f1[BIGKKNUTH-i].x.r)/((double) BIGKKNUTH);
}

power2 *= power2; /* exact */
for(i=0; i < BIGKKNUTH; ++i) {
    f2[i].x.l *= power2; /* exact */
    f2[i].x.r *= power2; /* exact */
}

for(i=0; i < BIGKKNUTH - 1; ++i) { /* normalize digits */
    while(f2[i].x.r >= 256.0) {
        f2[i+1].x = intsum(f2[i+1].x,ONE);
        f2[i].x = intdiff(f2[i].x,INT256);
    }
}

(void) printf("\
\n For the input values %d and %d\n\n",
    num0,num1);

```

```

product = num0 * num1;
for(i=3,pd = (unsigned char *) &product;
    i >= 0;
    --i,++pd) {
    (void) printf("\
Actual base-256 digit of product: %d\n\
[%20.16e %20.16e]\n[(%s) (%s)]\n",
        ((int) *pd),
        f2[i].x.l,f2[i].x.r,
        litprint(f2[i].x.l),litprint(f2[i].x.r));
    }
}

```

```

fft(pa)
struct complex pa[];
{
int i,j0,j1,k,index0,index1,pow20,pow21;
struct complex e0,e1,u,v;

for(i=1; i <= KKNUTH; ++i) {
    pow20 = 1<<i;
    pow21 = 1<<(KKNUTH-i);
    for(j0=0; j0 < pow20; j0+=2) {
        j1 = j0+1;
        e0 = wpow[reverse[j0]];
        e1 = wpow[reverse[j1]];
        for(k=0; k < pow21; ++k) {
            index0 = j0*pow21+k;
            index1 = index0 + pow21;
            u = pa[index0];
            v = pa[index1];
            pa[index0].x = intsum(
                u.x,
                intdiff(
                    intprod(e0.x,v.x),
                    intprod(e0.y,v.y)

```

```

        )
    );
    pa[index0].y = intsum(
        u.y,
        intsum(
            intprod(e0.x,v.y),
            intprod(e0.y,v.x)
        )
    );
    pa[index1].x = intsum(
        u.x,
        intdiff(
            intprod(e1.x,v.x),
            intprod(e1.y,v.y)
        )
    );
    pa[index1].y = intsum(
        u.y,
        intsum(
            intprod(e1.x,v.y),
            intprod(e1.y,v.x)
        )
    );
}
}
}
for(i=0; i < BIGKKNUTH; ++i) {
    k = reverse[i];
    if(i > k) {
        u = pa[i];
        pa[i] = pa[k];
        pa[k] = u;
    }
}
}

```

```

#include "intops.c"

```

C.2 VAX Interval FFT Multiply

The second program uses VAX double-precision floating-point arithmetic as implemented on a VAX 11/750. Since VAX arithmetic does not support different rounding modes, its interval operations produce upper and lower bounds on computed quantities by adding or subtracting amounts equal to the least significant mantissa bits of these quantities.

```
#include <stdio.h>
#include <math.h>

#define KKNUTH 8
#define LKNUTH 8
#define BIGKKNUTH 256

int reverse[BIGKKNUTH];

struct interval {double l,r;};
struct interval ZERO = {0.0, 0.0};
struct interval ONE = {1.0, 1.0};
struct interval TWO = {2.0,2.0};
struct interval INT256 = {256.0,256.0};

struct complex {struct interval x,y;} wpow[BIGKKNUTH],
               w2pow[KKNUTH+1];

main()
{
    char *litprint();
    unsigned char *pd;
    unsigned num0,num1,product;
    int i,j,k,rk;
    double power2;
    struct interval temp;
    struct interval intsum(),intdiff(),
    intprod(),intquot(),intsqrt();
```

```

struct complex s,t,f1[BIGKKNUTH],f2[BIGKKNUTH];

for(i=0; i < BIGKKNUTH; ++i) {
    k = i;
    rk = 0;
    for(j=1; j < KKNUTH; ++j) {
        if(k&1)
            rk += 1;
        k >>= 1;
        rk <<= 1;
    }
    if(k&1)
        rk += 1;
    reverse[i] = rk;
}

w2pow[1].x.l = -1.0;
w2pow[1].x.r = -1.0;
w2pow[1].y.l = 0.0;
w2pow[1].y.r = 0.0;
w2pow[2].x.l = 0.0;
w2pow[2].x.r = 0.0;
w2pow[2].y.l = 1.0;
w2pow[2].y.r = 1.0;
for(i=3; i <= KKNUTH; ++i) {
    w2pow[i].x = intsqrt(
        intquot(
            intsum(ONE,w2pow[i-1].x),
            TWO
        )
    );
    w2pow[i].y = intsqrt(
        intquot(
            intdiff(ONE,w2pow[i-1].x),
            TWO
        )
    );
}

```



```

    }
    w2pow[0] = w2pow[KKNUTH];

    for(i=0; i < BIGKKNUTH; ++i) {
        s.x = ONE;
        s.y = ZERO;
        j = i;
        k = 0;
        while(j != 0) {
            if(j&1) {
                t.x = intdiff(
                    intprod(s.x,w2pow[KKNUTH-k].x),
                    intprod(s.y,w2pow[KKNUTH-k].y)
                );
                t.y = intsum(
                    intprod(s.x,w2pow[KKNUTH-k].y),
                    intprod(s.y,w2pow[KKNUTH-k].x)
                );
                s = t;
            }
            j >>= 1;
            ++k;
        }
        wpow[i] = s;
    }

    (void) fprintf(stderr,"input first positive number: ");
    (void) scanf("%d",&num0);
    (void) fprintf(stderr,"input second positive number: ");
    (void) scanf("%d",&num1);

    power2 = (double) (1<<(KKNUTH + LKNUTH)); /* exact */
    for(i=0,pd = (unsigned char *) &num0;
        i < 4;
        ++i,++pd) {
        f1[i].x.l = ((double) *pd)/power2;    /* exact */
        f1[i].x.r = f1[i].x.l;
    }

```

```

        f1[i].y.l = 0.0;
        f1[i].y.r = 0.0;
    }
    for(i=4; i < BIGKKNUTH; ++i) {
        f1[i].x.l = 0.0;
        f1[i].x.r = 0.0;
        f1[i].y.l = 0.0;
        f1[i].y.r = 0.0;
    }
    for(i=0,pd = (unsigned char *) &num1;
        i < 4;
        ++i,++pd) {
        f2[i].x.l = ((double) *pd)/power2;    /* exact */
        f2[i].x.r = f2[i].x.l;
        f2[i].y.l = 0.0;
        f2[i].y.r = 0.0;
    }
    for(i=4; i < BIGKKNUTH; ++i) {
        f2[i].x.l = 0.0;
        f2[i].x.r = 0.0;
        f2[i].y.l = 0.0;
        f2[i].y.r = 0.0;
    }

    fft(f1);
    fft(f2);
    for(i=0; i < BIGKKNUTH; ++i) {
        temp = f1[i].x;
        f1[i].x = intdiff(
            intprod(temp,f2[i].x),
            intprod(f1[i].y,f2[i].y)
        );
        f1[i].y = intsum(
            intprod(temp,f2[i].y),
            intprod(f1[i].y,f2[i].x)
        );
    }

```

```

fft(f1);

f2[0].x.l = (f1[0].x.l)/((double) BIGKKNUTH); /* exact */
f2[0].x.r = (f1[0].x.r)/((double) BIGKKNUTH); /* exact */
for(i=1; i < BIGKKNUTH; ++i) {
    f2[i].x.l = (f1[BIGKKNUTH-i].x.l)/((double) BIGKKNUTH);
    f2[i].x.r = (f1[BIGKKNUTH-i].x.r)/((double) BIGKKNUTH);
}

power2 *= power2;                /* exact */
for(i=0; i < BIGKKNUTH; ++i) {
    f2[i].x.l *= power2;          /* exact */
    f2[i].x.r *= power2;          /* exact */
}

for(i=0; i < BIGKKNUTH - 1; ++i) { /* normalize digits */
    while(f2[i].x.r >= 256.0) {
        f2[i+1].x = intsum(f2[i+1].x,ONE);
        f2[i].x = intdiff(f2[i].x,INT256);
    }
}

(void) printf("\
\n For the input values %d and %d\n\n",num0,num1);
product = num0 * num1;
for(i=3,pd = ((unsigned char *) &product) + 3;
    i >= 0;
    --i,--pd) {
    (void) printf("\
Actual base-256 digit of product: %d\n\
[%20.16e %20.16e]\n[(%s) (%s)]\n",
    ((int) *pd),
    f2[i].x.l,f2[i].x.r,
    litprint(f2[i].x.l),litprint(f2[i].x.r));
}
}

```

```

fft(pa)
struct complex pa[];
{
int i,j0,j1,k,index0,index1,pow20,pow21;
struct complex e0,e1,u,v;

for(i=1; i <= KKNUTH; ++i) {
    pow20 = 1<<i;
    pow21 = 1<<(KKNUTH-i);
    for(j0=0; j0 < pow20; j0+=2) {
        j1 = j0+1;
        e0 = wpow[reverse[j0]];
        e1 = wpow[reverse[j1]];
        for(k=0; k < pow21; ++k) {
            index0 = j0*pow21+k;
            index1 = index0 + pow21;
            u = pa[index0];
            v = pa[index1];
            pa[index0].x = intsum(
                u.x,
                intdiff(
                    intprod(e0.x,v.x),
                    intprod(e0.y,v.y)
                )
            );
            pa[index0].y = intsum(
                u.y,
                intsum(
                    intprod(e0.x,v.y),
                    intprod(e0.y,v.x)
                )
            );
            pa[index1].x = intsum(
                u.x,
                intdiff(
                    intprod(e1.x,v.x),

```

```

                                intprod(e1.y,v.y)
                                )
                                );
    pa[index1].y = intsum(
                                u.y,
                                intsum(
                                    intprod(e1.x,v.y),
                                    intprod(e1.y,v.x)
                                )
                                );
    }
}
}
for(i=0; i < BIGKKNUTH; ++i) {
    k = reverse[i];
    if(i > k) {
        u = pa[i];
        pa[i] = pa[k];
        pa[k] = u;
    }
}
}

```

```

#define downsum(X,Y) decrement((X)+(Y))
#define upsum(X,Y) increment((X)+(Y))

struct interval intsum(int1,int2)
struct interval int1,int2;
{
    double decrement(),increment();
    struct interval intr;

    intr.l = downsum(int1.l,int2.l);
    intr.r = upsum(int1.r,int2.r);
    return(intr);
}

```

```

struct interval intdiff(int1,int2)
struct interval int1,int2;
{
double decrement(),increment();
struct interval intr;

intr.l = downsum(int1.l,-int2.r);
intr.r = upsum(int1.r,-int2.l);
return(intr);
}

#define downprod(X,Y) decrement((X)*(Y))
#define upprod(X,Y) increment((X)*(Y))

struct interval intprod(int1,int2)
struct interval int1,int2;
{
double temp1,temp2;
double decrement(),increment();
struct interval intr;

if(int1.l >= 0.0) {
    if(int2.l >= 0.0) {
        intr.l = downprod(int1.l,int2.l);
        intr.r = upprod(int1.r,int2.r);
    }
    else {
        if(int2.r <= 0.0) {
            intr.l = downprod(int1.r,int2.l);
            intr.r = upprod(int1.l,int2.r);
        }
        else {
            intr.l = downprod(int1.r,int2.l);
            intr.r = upprod(int1.r,int2.r);
        }
    }
}

```

```

    }
  }
}
else {
  if(int1.r <= 0.0) {
    if(int2.l >= 0.0) {
      intr.l = downprod(int1.l,int2.r);
      intr.r = upprod(int1.r,int2.l);
    }
    else {
      if(int2.r <= 0.0) {
        intr.l = downprod(int1.r,int2.r);
        intr.r = upprod(int1.l,int2.l);
      }
      else {
        intr.l = downprod(int1.l,int2.r);
        intr.r = upprod(int1.l,int2.l);
      }
    }
  }
}
else {
  if(int2.l >= 0.0) {
    intr.l = downprod(int1.l,int2.r);
    intr.r = upprod(int1.r,int2.r);
  }
  else {
    if(int2.r <= 0.0) {
      intr.l = downprod(int1.r,int2.l);
      intr.r = upprod(int1.l,int2.l);
    }
    else {
      temp1 = downprod(int1.l,int2.r);
      temp2 = downprod(int1.r,int2.l);
      intr.l = (temp1 <= temp2)?temp1:temp2;
      temp1 = upprod(int1.l,int2.l);
      temp2 = upprod(int1.r,int2.r);
      intr.r = (temp1 >= temp2)?temp1:temp2;
    }
  }
}

```

```

    }
  }
}
return(intr);
}

```

```

#define downquot(X,Y) decrement((X)/(Y))
#define upquot(X,Y) increment((X)/(Y))

```

```

struct interval intquot(int1,int2)
struct interval int1,int2;
{
double decrement(),increment();
struct interval intr;

```

```

if(int2.l <= 0.0 && int2.r >= 0.0) {
    (void) fprintf(stderr,"interval division by zero\n");
    exit(1);
}

```

```

if(int1.l >= 0.0) {
    if(int2.l > 0.0) {
        intr.l = downquot(int1.l,int2.r);
        intr.r = upquot(int1.r,int2.l);
    }
    else {
        intr.l = downquot(int1.r,int2.r);
        intr.r = upquot(int1.l,int2.l);
    }
}

```

```

else {
    if(int1.r <= 0.0) {
        if(int2.l > 0.0) {
            intr.l = downquot(int1.l,int2.l);
            intr.r = upquot(int1.r,int2.r);

```



```

    }
    else {
        intr.l = downquot(int1.r,int2.l);
        intr.r = upquot(int1.l,int2.r);
    }
}
else {
    if(int2.l > 0.0) {
        intr.l = downquot(int1.l,int2.l);
        intr.r = upquot(int1.r,int2.l);
    }
    else {
        intr.l = downquot(int1.r,int2.r);
        intr.r = upquot(int1.l,int2.r);
    }
}
}
return(intr);
}

```

```

struct interval intsqrt(intx)
struct interval intx;
{
double increment(),decrement();
struct interval result;

if(intx.l > intx.r || intx.r < 0.0) {
    (void) fprintf(stderr,"square root of negative quantity\n");
    exit(1);
}
else {
    if(intx.l < 0.0)
        result.l = 0.0;
    else
        result.l = sqrt(intx.l);
    result.r = sqrt(intx.r);
}
}

```

```

        if(result.l > 0.0)
            result.l = decrement(result.l);
        if(result.r > 0.0)
            result.r = increment(result.r);
    }
    return(result);
}

```

```

#define MAX 1.7014118346046923e+38
#define NEGMAX -1.7014118346046923e+38
#define MIN 2.9387358770557188e-39
#define NEGMIN -2.9387358770557188e-39

```

```

double increment(x)
double x;
{
    unsigned char *pexp10,*pexp11,*pexp20,*pexp21;
    double min1;

```

```

    if(x == MAX) {
        (void) fprintf(stderr, "\
attempt to increase largest value\n");
        exit(1);
    }

```

```

    if(x == 0.0)
        return(MIN);

```

```

    if(x == NEGMIN)
        return(0.0);

```

```

    min1 = 0.0;
    pexp10 = ((unsigned char *) &x);
    pexp11 = pexp10 + 1;
    pexp20 = ((unsigned char *) &min1);

```

```

pexp21 = pexp20 + 1;
*pexp20 = *pexp10 & 128; /* modifies min1 -- remember, */
*pexp21 = *pexp11 & 127; /* leading mantissa 1 implicit */
if(*pexp21 >= 28) {
    *pexp21 -= 28;          /* min1 is now half the value */
    x += 2.0 * min1;        /* of the least bit of x */
}
else {
    *pexp11 += 28;          /* scale x to avoid underflow */
    x += 2.0 * min1;        /* min1 again half least bit */
    *pexp11 -= 28;          /* scale x back */
}

return(x);
}

```

```

double decrement(x)
double x;
{
    unsigned char *pexp10,*pexp11,*pexp20,*pexp21;
    double min1;

    if(x == NEGMAX) {
        (void) fprintf(stderr,"\
attempt to decrease most negative value\n");
        exit(1);
    }

    if(x == 0.0)
        return(NEGMIN);

    if(x == MIN)
        return(0.0);

    min1 = 0.0;
    pexp10 = ((unsigned char *) &x);

```

```

pexp11 = pexp10 + 1;
pexp20 = ((unsigned char *) &min1);
pexp21 = pexp20 + 1;
*pexp20 = *pexp10 & 128; /* modifies min1 -- remember, */
*pexp21 = *pexp11 & 127; /* leading mantissa 1 implicit */
if(*pexp21 >= 28) {
    *pexp21 -= 28;          /* min1 is now half the value */
    x -= 2.0 * min1;        /* of the least bit of x */
}
else {
    *pexp11 += 28;          /* scale x to avoid underflow */
    x -= 2.0 * min1;        /* min1 again half least bit */
    *pexp11 -= 28;          /* scale x back */
}

return(x);
}

```

```

/* LITMAX should be as large as the largest number of */
/* calls to litprint in a single invocation of printf */

```

```

#define LITMAX 10
#define LITLNGTH 20

```

```

char *litprint(x)
double x;
{
    static char litstrings[LITMAX][LITLNGTH];
    static int litindex = 0; /* compiler init needed */
    char *pstring,*sprintf();
    unsigned short *px;

```

```

    px = (unsigned short *) &x;
    pstring = litstrings[litindex];

```

```

    (void) sprintf(pstring,"%04x %04x %04x %04x",

```

```

        *px,* (px+1),*(px+2),*(px+3));

    ++litindex;
    if(litindex == LITMAX)
        litindex = 0;

    return(pstring);
}

```

C.3 Scalar FFT Multiply

The final program implements an ordinary floating-point version of an algorithm to use Fast Fourier Transforms to multiply large integers. Although the program as it is given is written to run on a Sun, comments in the code describe the simple adaptations needed to have it produce similar results on the VAX.

```

#include <stdio.h>
#include <math.h>

#define KKNUTH 8
#define LKNUTH 8
#define BIGKKNUTH 256

int reverse[BIGKKNUTH];
struct complex {double x,y;} wpow[BIGKKNUTH],
                    w2pow[KKNUTH+1];

main()
{
    char *litprint();
    unsigned char *pd;
    unsigned num0,num1,product;
    int i,j,k,rk;
    double power2,temp;

```

```

struct complex s,t,f1[BIGKKNUTH],f2[BIGKKNUTH];

for(i=0; i < BIGKKNUTH; ++i) {
    k = i;
    rk = 0;
    for(j=1; j < KKNUTH; ++j) {
        if(k&1)
            rk += 1;
        k >>= 1;
        rk <<= 1;
    }
    if(k&1)
        rk += 1;
    reverse[i] = rk;
}

w2pow[1].x = -1.0;
w2pow[1].y = 0.0;
w2pow[2].x = 0.0;
w2pow[2].y = 1.0;
for(i=3; i <= KKNUTH; ++i) {
    w2pow[i].x = sqrt((1.0 + w2pow[i-1].x)/2.0);
    w2pow[i].y = sqrt((1.0 - w2pow[i-1].x)/2.0);
}
w2pow[0] = w2pow[KKNUTH];

for(i=0; i < BIGKKNUTH; ++i) {
    s.x = 1.0;
    s.y = 0.0;
    j = i;
    k = 0;
    while(j != 0) {
        if(j&1) {
            t.x = s.x * w2pow[KKNUTH-k].x -
                s.y * w2pow[KKNUTH-k].y;
            t.y = s.x * w2pow[KKNUTH-k].y +
                s.y * w2pow[KKNUTH-k].x;

```

```

        s = t;
    }
    j >>= 1;
    ++k;
}
wpow[i] = s;
}

(void) fprintf(stderr,"input first positive number: ");
(void) scanf("%d",&num0);
(void) fprintf(stderr,"input second positive number: ");
(void) scanf("%d",&num1);

power2 = (double) (1<<(KKNUTH + LKNUTH));
for(i=0,pd = ((unsigned char *) &num0) + 3; /* See Note */
    i < 4;
    ++i,--pd) {
    /* See Note */
    f1[i].x = ((double) *pd)/power2;
    f1[i].y = 0.0;
}
for(i=4; i < BIGKKNUTH; ++i) {
    f1[i].x = 0.0;
    f1[i].y = 0.0;
}
for(i=0,pd = ((unsigned char *) &num1) + 3; /* See Note */
    i < 4;
    ++i,--pd) {
    /* See Note */
    f2[i].x = ((double) *pd)/power2;
    f2[i].y = 0.0;
}
for(i=4; i < BIGKKNUTH; ++i) {
    f2[i].x = 0.0;
    f2[i].y = 0.0;
}

fft(f1);
fft(f2);

```

```

for(i=0; i < BIGKKNUTH; ++i) {
    temp = f1[i].x;
    f1[i].x = temp * f2[i].x - f1[i].y * f2[i].y;
    f1[i].y = temp * f2[i].y + f1[i].y * f2[i].x;
}
fft(f1);

f2[0].x = (f1[0].x)/((double) BIGKKNUTH);
for(i=1; i < BIGKKNUTH; ++i)
    f2[i].x = (f1[BIGKKNUTH-i].x)/((double) BIGKKNUTH);

power2 *= power2;
for(i=0; i < BIGKKNUTH; ++i)
    f2[i].x *= power2;

for(i=0; i < BIGKKNUTH - 1; ++i) { /* normalize */
    while(f2[i].x >= 256.0) {
        f2[i+1].x += 1.0;
        f2[i].x -= 256.0;
    }
}

(void) printf("\
\n  For the input values %d and %d\n\n",
    num0,num1);

product = num0 * num1;
for(i=3,pd = (unsigned char *) &product; /* See Note */
    i >= 0;
    --i,++pd) { /* See Note */
    (void) printf("\
Actual base-256 digit of product: %d\n\
%20.16e\n(%s)\n",
        ((int) *pd),f2[i].x,litprint(f2[i].x));
    }
}

```


/* Note: On the VAX, the individual bytes of a floating point value are addressed differently. The three loops noted above should be changed to

```
for(i=0,pd = ((unsigned char *) &num0);
    i < 4;
    ++i,++pd) {

for(i=0,pd = ((unsigned char *) &num1);
    i < 4;
    ++i,++pd) {

for(i=3,pd = (unsigned char *) &product + 3;
    i >= 0;
    --i,--pd) {
```

to make the program behave the same on the VAX as it does on the Suns. */

```
fft(pa)
struct complex pa[];
{
int i,j0,j1,k,index0,index1,pow20,pow21;
struct complex e0,e1,u,v;

for(i=1; i <= KKNUTH; ++i) {
    pow20 = 1<<i;
    pow21 = 1<<(KKNUTH-i);
    for(j0=0; j0 < pow20; j0+=2) {
        j1 = j0+1;
        e0 = wpow[reverse[j0]];
        e1 = wpow[reverse[j1]];
        for(k=0; k < pow21; ++k) {
            index0 = j0*pow21+k;
            index1 = index0 + pow21;
```

```

        u = pa[index0];
        v = pa[index1];
        pa[index0].x = u.x + e0.x*v.x - e0.y*v.y;
        pa[index0].y = u.y + e0.x*v.y + e0.y*v.x;
        pa[index1].x = u.x + e1.x*v.x - e1.y*v.y;
        pa[index1].y = u.y + e1.x*v.y + e1.y*v.x;
    }
}

for(i=0; i < BIGKKNUTH; ++i) {
    k = reverse[i];
    if(i > k) {
        u = pa[i];
        pa[i] = pa[k];
        pa[k] = u;
    }
}

/* LITMAX should be as large as the largest number of */
/* calls to litprint in a single invocation of printf */

#define LITMAX 10
#define LITLNGTH 20

char *litprint(x)
double x;
{
    static char litstrings[LITMAX][LITLNGTH];
    static int litindex = 0; /* compiler init needed */
    char *pstring,*sprintf();
    unsigned short *px;

    px = (unsigned short *) &x;
    pstring = litstrings[litindex];

```

```
(void) sprintf(pstring,"%04x %04x %04x %04x",
               *px,*px+1,*px+2,*px+3));

++litindex;
if(litindex == LITMAX)
    litindex = 0;

return(pstring);
}
```

Appendix D

FFT Multiply Comparisons

This appendix contains an edited collection of output from the interval and scalar versions of the programs given in Appendix C, programs that use Fast Fourier Transforms to multiply large integers. In all cases, results are first given in decimal form, then as hexadecimal descriptions of the actual bit patterns that code the double-precision interval endpoints or scalar values on the machine being used. These results are discussed in Chapters 2 and 3.

The Sun interval results were obtained using upwardly- or downwardly-rounded IEEE double-precision floating-point values, as implemented on a Sun 3/60 having an MC68881 coprocessor with mask A93N, as interval endpoints. The VAX interval results were obtained by forming upper and lower bounds on computed quantities by adding or subtracting amounts equal to the least significant mantissa bits of these computed quantities. The VAX interval computations were made on a VAX 11/750. The Sun scalar results were obtained using the IEEE default, round-to-nearest rounding mode on the same Sun 3/60 machine. The VAX scalar results were obtained using the only rounding mode available to VAX arithmetic, which is similar to round-to-nearest but rounds to the value with larger magnitude rather than the value with least significant bit 0 when two representable values are equally close to a nonrepresentable intermediate result. The VAX scalar computations were made on the same VAX 11/750.

The edited output begins on the next page.

For the input values 17 and 34

Actual base-256 digit of product: 0

```
[0.0000000000000000e+00 0.0000000000000000e+00] (IEEE)
[(8000 0000 0000 0000) (0000 0000 0000 0000)]
[-3.0804118409741014e-13 3.0804118409741014e-13] (VAX)
[(abad 6969 c640 0b7b) (2bad 6969 c640 0b7b)]
0.0000000000000000e+00 (IEEE)
(0000 0000 0000 0000)
0.0000000000000000e+00 (VAX)
(0000 0000 0000 0000)
```

Actual base-256 digit of product: 0

```
[0.0000000000000000e+00 0.0000000000000000e+00] (IEEE)
[(8000 0000 0000 0000) (0000 0000 0000 0000)]
[-2.6524986067235032e-13 2.6524986067235032e-13] (VAX)
[(ab95 5288 973f f980) (2b95 5288 973f f980)]
0.0000000000000000e+00 (IEEE)
(0000 0000 0000 0000)
0.0000000000000000e+00 (VAX)
(0000 0000 0000 0000)
```

Actual base-256 digit of product: 2

```
[2.0000000000000000e+00 2.0000000000000000e+00] (IEEE)
[(4000 0000 0000 0000) (4000 0000 0000 0000)]
[1.999999999997548e+00 2.0000000000002452e+00] (VAX)
[(40ff ffff ffff dd7f) (4100 0000 0000 1141)]
2.0000000000000000e+00 (IEEE)
(4000 0000 0000 0000)
2.0000000000000000e+00 (VAX)
(4100 0000 0000 0000)
```

Actual base-256 digit of product: 66

```

[6.6000000000000000e+01 6.6000000000000000e+01] (IEEE)
[(4050 8000 0000 0000) (4050 8000 0000 0000)]
[6.5999999999999994e+01 6.6000000000000005e+01] (VAX)
[(4383 ffff ffff fee3) (4384 0000 0000 011d)]
6.6000000000000000e+01 (IEEE)
(4050 8000 0000 0000)
6.6000000000000000e+01 (VAX)
(4384 0000 0000 0000)

```

For the input values 8724 and 9683

Actual base-256 digit of product: 5

```

[4.9999999999685540e+00 5.0000000000314415e+00] (IEEE)
[(4013 ffff ffff 75b3) (4014 0000 0000 8a48)]
[4.9999999999822584e+00 5.0000000000177497e+00] (VAX)
[(419f ffff fffd 8fc6) (41a0 0000 0002 7083)]
4.9999999999999734e+00 (IEEE)
(4013 ffff ffff ffe2)
5.0000000000000000e+00 (VAX)
(41a0 0000 0000 0000)

```

Actual base-256 digit of product: 8

```

[7.9999999999729425e+00 8.0000000000275122e+00] (IEEE)
[(401f ffff ffff 8900) (4020 0000 0000 3c80)]
[7.9999999999841335e+00 8.0000000000161793e+00] (VAX)
[(41ff ffff fffd d1bf) (4200 0000 0001 1ca1)]
8.00000000000002274e+00 (IEEE)
(4020 0000 0000 0080)
8.00000000000001705e+00 (VAX)
(4200 0000 0000 0300)

```

Actual base-256 digit of product: 250

[2.499999999994088e+02 2.5000000000006276e+02] (IEEE)
 [(406f 3fff ffff f7e0) (406f 4000 0000 08a0)]
 [2.499999999996529e+02 2.5000000000003619e+02] (VAX)
 [(4479 ffff ffff d9d5) (447a 0000 0000 27cb)]
 2.500000000000000e+02 (IEEE)
 (406f 4000 0000 0000)
 2.5000000000000091e+02 (VAX)
 (447a 0000 0000 0100)

Actual base-256 digit of product: 124

[1.239999999998454e+02 1.2400000000001546e+02] (IEEE)
 [(405e ffff ffff fbc0) (405f 0000 0000 0440)]
 [1.239999999999007e+02 1.2400000000001038e+02] (VAX)
 [(43f7 ffff ffff ea2b) (43f8 0000 0000 16d5)]
 1.240000000000000e+02 (IEEE)
 (405f 0000 0000 0000)
 1.2400000000000011e+02 (VAX)
 (43f8 0000 0000 0040)

For the input values 9473281 and 6734529

Actual base-256 digit of product: 38

[3.7999999999447027e+01 3.80000000000574801e+01] (IEEE)
 [(4042 ffff fffe d000) (4043 0000 0001 3c00)]
 [3.7999999999444603e+01 3.80000000000564492e+01] (VAX)
 [(4317 ffff fff6 7557) (4318 0000 0009 b2a9)]
 3.8000000000007276e+01 (IEEE)
 (4043 0000 0000 0400)
 3.8000000000006366e+01 (VAX)
 (4318 0000 0000 1c00)

Actual base-256 digit of product: 59

[5.8999999999548891e+01 5.9000000000465661e+01] (IEEE)

```

[(404d 7fff ffff 0800) (404d 8000 0001 0000)]
[5.8999999999541919e+01 5.9000000000467176e+01] (VAX)
[(436b ffff fff8 2157) (436c 0000 0008 06a9)]
5.9000000000000000e+01 (IEEE)
(404d 8000 0000 0000)
5.90000000000006366e+01 (VAX)
(436c 0000 0000 1c00)

```

Actual base-256 digit of product: 15

```

[1.4999999999570719e+01 1.5000000000440195e+01] (IEEE)
[(402d ffff fffc 5000) (402e 0000 0003 c800)]
[1.4999999999748830e+01 1.5000000000256172e+01] (VAX)
[(426f ffff ffee bd5f) (4270 0000 0011 9aa1)]
1.5000000000003638e+01 (IEEE)
(402e 0000 0000 0800)
1.5000000000003183e+01 (VAX)
(4270 0000 0000 3800)

```

Actual base-256 digit of product: 193

```

[1.929999999982782e+02 1.9300000000017249e+02] (IEEE)
[(4068 1fff ffff e856) (4068 2000 0000 17b5)]
[1.929999999990635e+02 1.9300000000009354e+02] (VAX)
[(4440 ffff ffff 9907) (4441 0000 0000 66d8)]
1.929999999999977e+02 (IEEE)
(4068 1fff ffff fff8)
1.9300000000000008e+02 (VAX)
(4441 0000 0000 0017)

```


Appendix E

Interval-Newton's Code

The following code implements Alefeld and Herzberger's [Ale86] Interval Newton's Method, discussed in Chapter 2, for IEEE double-precision floating-point arithmetic as implemented on a Sun 3/60 having an MC68881 coprocessor with mask A93N. Its interval operations force results to be rounded correctly by calling `fpmode_` under Release 3.5 of the Sun UNIX 4.2 operating system. Its interval operations, obtained from the file `intops.c`, are given in Appendix B; their semantics is specified in Appendix A.

```
#include <stdio.h>
#include <math.h>

unsigned oldmode,newmode,fpmode_();

double PLUINF,MININF;

struct interval {double l,r;};
struct interval EMPTY = {1.0, 0.0};
struct interval TWO = {2.0, 2.0};
struct interval POSINF,NEGINF;

#define MAXDEG 10

int degree;
```

```

double coeff[MAXDEG+1];

void main()
{
void doinits();
char *litprint();
int i;
struct interval intlength(),midpoint(),intinter();
struct interval intdiff(),intquot();
struct interval intf();
struct interval start,intx,intm,middle,newlength,oldlength;

doinits();

(void) fprintf(stderr,"input degree <= %d of polynomial f: ",
MAXDEG);
(void) scanf("%d",&degree);
for(i=degree; i >= 0; --i) {
(void) fprintf(stderr,"input coefficient of x^%d: ",i);
(void) scanf("%lf",&coeff[i]);
}
(void) fprintf(stderr,"input point x at which f(x) < 0: ");
(void) scanf("%lf",&start.l);
(void) fprintf(stderr,"input point x at which f(x) > 0: ");
(void) scanf("%lf",&start.r);
(void) fprintf(stderr,
"For the next two inputs, x ranges over the interval\n\
just input, and r is a root of f in this interval.\n");
(void) fprintf(stderr,
"input positive lower bound on f(x)/(x-r): ");
(void) scanf("%lf",&intm.l);
(void) fprintf(stderr,
"input positive upper bound on f(x)/(x-r): ");
(void) scanf("%lf",&intm.r);

intx = start;
newlength = intlength(intx);

```

```

do {
    oldlength = newlength;
    middle = midpoint(intx);
    intx = intinter(
        intx,
        intdiff(
            middle,
            intquot(
                intf(middle),
                intm
            )
        )
    );
    newlength = intlength(intx);
} while(newlength.r < oldlength.l);

(void) printf("\n  For the polynomial f(x) = \n");
for(i=degree; i >= 0; --i) {
    (void) printf("      %20.16e x^%d",coeff[i],i);
    if(i > 0)
        (void) printf(" +\n");
    else
        (void) printf(" ,\n");
}
(void) printf("  if the interval\n\
    [%20.16e %20.16e]\n\
contains a root r, and for every x in the\n\
interval it is true that f(x)/(x-r) belongs\n\
to the interval\n\
    [%20.16e %20.16e],\n\
then a root of f is contained in the interval:\n\n\
    [%20.16e %20.16e], i.e.,\n\
    [(%s) (%s)].\n\n",
        start.l,start.r,
        intm.l,intm.r,
        intx.l,intx.r,
        litprint(intx.l),litprint(intx.r));

```

```

    }

    struct interval intf(intx)
    struct interval intx;
    {
    double low,high,f();
    struct interval result;

    newmode = 2*64 + 2*16;
    oldmode = fpmode_(&newmode);
    low = f(intx.l);
    high = f(intx.r);
    result.l = (low <= high)?low:high;
    newmode = 2*64 + 3*16;
    newmode = fpmode_(&newmode);
    low = f(intx.l);
    high = f(intx.r);
    result.r = (high >= low)?high:low;
    newmode = fpmode_(&oldmode);

    return(result);
    }

    double f(x)
    double x;
    {
    int i;
    double y;

    y = 0.0;
    for(i=degree; i >= 0; --i) {
        y *= x;
        y += coeff[i];
    }
    return(y);
    }

```

```

struct interval midpoint(intx)
struct interval intx;
{
struct interval leftend(),rightend(),
intsum(),intquot();
struct interval result;

result = intquot(
    intsum(
        leftend(intx),
        rightend(intx)
    ),
    TWO
);

return(result);
}

#include "intops.c"

```

Appendix F

Interval-Newton's Results

This appendix contains output results from the program implementing the interval version of Newton's Method developed by Alefeld and Herzberger [Ale86] and given in Appendix E. All calculations were performed on a Sun 3/60 having an MC68881 coprocessor with mask A93N. All calculations were performed in the IEEE default round-to-nearest rounding mode. These results are discussed in Subsection 2.3.2.

By way of comparison, the arbitrary-precision constructive-real calculator developed by Boehm [Boe87] gives the following:

$$\begin{aligned}\sqrt{2} &\approx 1.414213562373095048801688724, \\ \sqrt[3]{3} &\approx 1.442249570307408382321638311, \text{ and} \\ \sqrt[5]{5} &\approx 1.379729661461214832390063464.\end{aligned}$$

```
For the polynomial f(x) =
  1.0000000000000000e+00 x^2 +
  0.0000000000000000e+00 x^1 +
  -2.0000000000000000e+00 x^0 ,
if the interval
  [1.0000000000000000e+00 2.0000000000000000e+00]
contains a root r, and for every x in the
interval it is true that f(x)/(x-r) belongs
to the interval
```

[2.0000000000000000e+00 4.0000000000000000e+00],
then a root of f is contained in the interval:

[1.4142135623730949e+00 1.4142135623730951e+00], i.e.,
[(3ff6 a09e 667f 3bcc) (3ff6 a09e 667f 3bcd)].

For the polynomial $f(x) =$

1.0000000000000000e+00 x^3 +
0.0000000000000000e+00 x^2 +
0.0000000000000000e+00 x^1 +
-3.0000000000000000e+00 x^0 ,

if the interval

[1.0000000000000000e+00 2.0000000000000000e+00]

contains a root r , and for every x in the
interval it is true that $f(x)/(x-r)$ belongs
to the interval

[3.0000000000000000e+00 1.2000000000000000e+01],

then a root of f is contained in the interval:

[1.4422495703074083e+00 1.4422495703074085e+00], i.e.,
[(3ff7 1374 4912 3ef6) (3ff7 1374 4912 3ef7)].

For the polynomial $f(x) =$

1.0000000000000000e+00 x^5 +
0.0000000000000000e+00 x^4 +
0.0000000000000000e+00 x^3 +
0.0000000000000000e+00 x^2 +
0.0000000000000000e+00 x^1 +
-5.0000000000000000e+00 x^0 ,

if the interval

[1.0000000000000000e+00 2.0000000000000000e+00]

contains a root r , and for every x in the
interval it is true that $f(x)/(x-r)$ belongs
to the interval

[5.0000000000000000e+00 8.0000000000000000e+01],

then a root of f is contained in the interval:

$[1.3797296614612147e+00 \ 1.3797296614612149e+00]$, i.e.,
 $[(3ff6 \ 135f \ 68d4 \ c0cb) \ (3ff6 \ 135f \ 68d4 \ c0cc)]$.

Appendix G

A Correctness Difficulty

As we explain in Section 2.4, the following C program is a counterexample to the spirit of our conjecture that programs using interval arithmetic that are provably asymptotically correct are also effectively asymptotically correct. The program computes π using Machin's formula [BB87],

$$\frac{\pi}{4} = 4 \arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right),$$

and the power series

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots.$$

```
#include <stdio.h>

main()
{
    double m,pow5,pow239;
    double low,high,oldlow,oldhigh;

    low = 0.0;
    high = 16.0/5.0 - 4.0/239.0;
    m = 1.0;
    pow5 = 5.0;
```

```

pow239 = 239.0;
do {
    oldlow = low;
    m += 2.0;
    pow5 *= 25.0;
    pow239 *= 57121.0;
    low = high - 16.0/(pow5*m) + 4.0/(pow239*m);

    oldhigh = high;
    m += 2.0;
    pow5 *= 25.0;
    pow239 *= 57121.0;
    high = low + 16.0/(pow5*m) - 4.0/(pow239*m);

} while(low < high && low > oldlow && high < oldhigh);

(void) printf(
"\n  Computed approximations to bounds on pi:\n\n\
  lower bound -- %20.16e\n\
  upper bound -- %20.16e\n\n",low,high);
}

```

Appendix H

Caliban Continued Fractions

The following Caliban code [BMS89], discussed in Subsection 6.4.1, computes and displays standard and generalized continued fractions. If the user simplifies the expression `getcfrac n` in the Clio prover [BMS89], where n is a base-10 expression for a nonnegative integer no greater than 134217728, the code produces a list of the fractions i/n for $n \leq i \leq 2n$ and their standard and optimal continued fraction expansions. The name `arith` refers to the file `arith.def`, which contains definitions of standard constants, arithmetic operations and order relations on natural numbers, integers and rationals. The file `arith.def` is given below.

Comments in the code use “term” to mean “partial quotient”. The `cfrac` function computes standard continued fractions and the `negfrac` function computes optimum ones. The name `negfrac` refers to the possible occurrence of negative numbers as partial quotients in optimum continued fractions.

```
FROM arith IMPORT izero,ione,illesseq,iabs
FROM arith IMPORT iplus,idiff,imult,idiv

|| Operations on integers

nextterm p q = t, illesseq (iabs (idiff p (imult t q)))
                (iabs (idiff p (imult tp q))); tp
  where t = idiv p q
        tp = iplus t ione, illesseq izero t; idiff t ione
```

```

cfrac <<p, <<ZERO,ZERO>> >> = []
cfrac <<p,q>> = t1 : (cfrac <<q, (idiff p (imult t1 q))>>)
      where t1 = idiv p q

negfrac <<p, <<ZERO,ZERO>> >> = []
negfrac <<p,q>> = t2 : (negfrac <<q, (idiff p (imult t2 q))>>)
      where t2 = nextterm p q

|| Operations creating continued fractions. They use NUMs
|| to enumerate the basic possibilities.

numseq a b = [], b<a; a:(numseq (a+1) b)

numtoint a = <<(#a),ZERO>>

lpair [] a = []
lpair (a:1) b = <<a,b>>:(lpair 1 b)

lcfrac [] = []
lcfrac (<<a,b>>:1) = lcfrac 1,cf1=cf2;
      << <<a,b>>,"\\n"),cf1,"\\n"),cf2,"\\n") >>
      : lcfrac 1
      where
      cf1 = (cfrac <<numtoint a, numtoint b>>)
      cf2 = (negfrac <<numtoint a, numtoint b>>)

getcfrac n = [],n<1; lcfrac (lpair (numseq n (2*n)) n)

```

The file `arith.def` follows. Its PROVE statements state mathematical facts that allow Clio to simplify many expressions.

```

|| Edited version of ~howard/testdir/arith.def
|| Also includes code from ~mark/oracled/testdir/nat.def

|| Define the natural numbers.

```

```

nplus ZERO n = n
nplus (SUCC n) m = SUCC (nplus n m)

ndiff n ZERO = n
ndiff ZERO n = ZERO
ndiff (SUCC m) (SUCC n) = ndiff m n

nmult ZERO n = (!n)->ZERO;bottom
nmult (SUCC m) n = nplus (nmult m n) n

nless ZERO (SUCC m) = true
nless n ZERO = false
nless (SUCC n) (SUCC m) = nless n m

nlesseq n n = true
nlesseq n m = nless n m

ndiv m ZERO = bottom
ndiv m n = (nlesseq (SUCC m) n)->ZERO;
           SUCC (ndiv (ndiff m n) n)

|| Define the integers. Integers are pairs of natural
|| numbers: <<m,n>> is m-n. In all except intermediate
|| calculations, at least one of m and n is ZERO.

isimp <<ZERO,n>> = <<ZERO,n>>
isimp <<m,ZERO>> = <<m,ZERO>>
isimp <<(SUCC m),(SUCC n)>> = isimp <<m,n>>

izero = <<ZERO,ZERO>>

ione = <<(#1),ZERO>>

iplus <<i,j>> <<k,l>> = isimp <<nplus i k, nplus j l>>
idiff <<i,j>> <<k,l>> = isimp <<nplus i l, nplus j k>>

```

```

imult <<i,j>> <<k,l>> =
    isimp <<(nplus (nmult i k) (nmult j l)),
        (nplus (nmult i l) (nmult j k))>>

```

```

idiv <<i,j>> <<k,k>> = bottom
idiv <<i,ZERO>> <<k,ZERO>> = <<ndiv i k, ZERO>>
idiv <<ZERO,l>> <<ZERO,l>> = <<ndiv j l, ZERO>>
idiv <<i,ZERO>> <<ZERO,l>> = <<ZERO, ndiv i l>>
idiv <<ZERO,j>> <<k,ZERO>> = <<ZERO, ndiv j k>>
idiv x y = idiv (isimp x) (isimp y)

```

```

iabs <<i,ZERO>> = <<i,ZERO>>
iabs <<ZERO,j>> = <<j,ZERO>>

```

|| Define relations on integers

```

illesseq <<i,j>> <<k,l>> = nlesseq (nplus i l) (nplus j k)

```

```

iter ZERO f s = s
iter (SUCC n) f s = iter n f (f s)

```

PROVE

```

'x=(SUCC x)' = '¬(!x)'

```

PROVE

```

'nplus ZERO n' = 'n'

```

PROVE

```

'nplus (SUCC n) m' = 'SUCC (nplus n m)'

```

PROVE

```

'nplus m n' = 'nplus n m'

```

PROVE

```

'nplus (nplus l m) n' = 'nplus l (nplus m n)'

```

PROVE

```

'ndiff n ZERO' = 'n'

```

PROVE

```

'ndiff (SUCC n) (SUCC m)' = 'ndiff n m'

```

PROVE

```

    'ndiff n n' = 'ZERO' , '!n'='true'
PROVE
    'ndiff (ndiff 1 m) n' = 'ndiff 1 (nplus m n)'
PROVE
    'nless n ZERO' = '~!n'
PROVE
    'nless (SUCC n) (SUCC m)' = 'nless n m'
PROVE
    'nless n (SUCC m)' = 'true',
        ('n=m'='true' \ / 'nless n m'='true')
PROVE
    'nless (SUCC n) m' = 'true',
        ('nless n 1'='true' & 'nless 1 m'='true')
PROVE
    'nless ZERO (SUCC m)' = 'true'
PROVE
    'nless n ZERO' = 'false', '!n' = 'true'
PROVE
    'nless (SUCC n) (SUCC m)' = 'nless n m'
PROVE
    'ndiv m ZERO' = 'bottom'
PROVE
    'ndiv m n' = ' (nlesseq (SUCC m) n)->ZERO;
        SUCC (ndiv (ndiff m n) n)'
PROVE
    'nmult ZERO n' = ' (!n)->ZERO;bottom'
PROVE
    'nmult (SUCC m) n' = ' nplus (nmult m n) n'

```

Appendix I

Gosper's Algorithm Code

This appendix contains C programs carrying out Gosper's algorithm to compute combinations of the number $1 + \sqrt{2} = [2, 2, 2, \dots]$ with itself, and an example showing how subroutines of these programs can be rewritten to make the programs compute combinations of other continued fractions. The first program computes standard continued fractions and uses a decision cube for deciding whether it is possible to determine and output the next partial quotient of the result, as proposed by Matula and Kornerup in [KM88]. The second program computes generalized continued fractions and uses a cruder algorithm to decide whether it can produce output. These programs and their outputs are discussed in Subsection 6.4.2. Both programs use IEEE double-precision floating-point arithmetic as implemented on a Sun 3/60 having an MC68881 coprocessor with mask A93N. They both clear and test the `inexact` status flag by calling `fpstatus_` under Release 3.5 of the Sun UNIX 4.2 operating system.

I.1 Standard Continued Fractions

```
#include <stdio.h>
```

```
double a,b,c,d,e,f,g,h,A,B,C,D,E,F,G,H;
```



```

main()
{
    unsigned oldstatus,newstatus,fpstatus_();
    double t,ta,tb,tc,td,te,tf,tg,th;
    double x(),y(),floor();
    double vx,vy,vz,p,q,pm1,qm1,pm2,qm2,rx(),ry(),rz();

    newstatus = 0;    /* for clearing program status flags */

    (void) fprintf(stderr,"initialize coefficient cube:\n");
    (void) fprintf(stderr,"a: ");
    (void) scanf("%lf",&a);
    (void) fprintf(stderr,"b: ");
    (void) scanf("%lf",&b);
    (void) fprintf(stderr,"c: ");
    (void) scanf("%lf",&c);
    (void) fprintf(stderr,"d: ");
    (void) scanf("%lf",&d);
    (void) fprintf(stderr,"e: ");
    (void) scanf("%lf",&e);
    (void) fprintf(stderr,"f: ");
    (void) scanf("%lf",&f);
    (void) fprintf(stderr,"g: ");
    (void) scanf("%lf",&g);
    (void) fprintf(stderr,"h: ");
    (void) scanf("%lf",&h);

    vx = rx();        /* prepare values for descriptive output */
    vy = ry();
    vz = rz(vx,vy);

    A = a;            /* initialize decision values */
    B = a + b;
    C = a + c;
    D = a + b + c + d;
    E = e;
    F = e + f;

```

```

G = e + g;
H = e + f + g + h;

pm2 = 0;      /* initialize convergents */
qm2 = 1;
pm1 = 1;
qm1 = 0;

while(1) {
    if(H == 0 || F == 0 ||
        floor(B/F) != floor(D/H)) { /* must ingest an x */

        t = x();

        oldstatus = fpstatus_(&newstatus); /* clear flags */

        ta = a;      /* update coefficient values */
        tb = b;
        tc = c;
        td = d;
        te = e;
        tf = f;
        tg = g;
        th = h;

        a = t*ta + tc;
        b = t*tb + td;
        c = ta;
        d = tb;
        e = t*te + tg;
        f = t*tf + th;
        g = te;
        h = tf;

        ta = A;      /* update decision values */
        tb = B;
        tc = C;

```

```

td = D;
te = E;
tf = F;
tg = G;
th = H;

C = t*ta + tc;
D = t*tb + td;
A = C - ta;
B = D - tb;
G = t*te + tg;
H = t*tf + th;
E = G - te;
F = H - tf;

oldstatus = fpstatus_(&newstatus); /* test flags */
if(oldstatus & 512) {
    (void) printf("\n\
Inexact update ingesting x partial quotient %1.0f\n",
t);
    exit(1);
}
(void) printf("\
Ingested the x partial quotient %1.0f\n",t);
}
else {
    if(G == 0 || E == 0 ||
        floor(C/G) != floor(D/H) ||
        floor(A/E) != floor(B/F)) { /* must ingest a y */

        t = y();

        oldstatus = fpstatus_(&newstatus);

        ta = a;      /* update coefficient values */
        tb = b;
        tc = c;

```

```

td = d;
te = e;
tf = f;
tg = g;
th = h;

a = t*ta + tb;
b = ta;
c = t*tc + td;
d = tc;
e = t*te + tf;
f = te;
g = t*tg + th;
h = tg;

ta = A;      /* update decision values */
tb = B;
tc = C;
td = D;
te = E;
tf = F;
tg = G;
th = H;

B = t*ta + tb;
A = B - ta;
D = t*tc + td;
C = D - tc;
F = t*te + tf;
E = F - te;
H = t*tg + th;
G = H - tg;

oldstatus = fpstatus_(&newstatus);
if(oldstatus & 512) {
    (void) printf("\n\
Inexact update ingesting y partial quotient %1.0f\n",

```

```

        t);
        exit(1);
    }
    (void) printf("\
Ingested the y partial quotient %1.0f\n",t);
}
else { /* can output a partial quotient */
    t = floor(A/E);

    oldstatus = fpstatus_(&newstatus);

    ta = a;      /* update coefficient values */
    tb = b;
    tc = c;
    td = d;
    te = e;
    tf = f;
    tg = g;
    th = h;

    a = te;
    b = tf;
    c = tg;
    d = th;
    e = ta - t*te;
    f = tb - t*tf;
    g = tc - t*tg;
    h = td - t*th;

    ta = A;      /* update decision values */
    tb = B;
    tc = C;
    td = D;
    te = E;
    tf = F;
    tg = G;
    th = H;

```

```

A = te;
B = tf;
C = tg;
D = th;
E = ta - t*te;
F = tb - t*tf;
G = tc - t*tg;
H = td - t*th;

oldstatus = fpstatus_(&newstatus);
if(oldstatus & 512) {
    (void) printf("\n\
Inexact update outputting partial quotient %1.0f\n",
t);
    exit(1);
}

(void) printf("\n\
Output the partial quotient %1.0f\n\n",
t);

(void) printf("\
Coefficient cube after output:\n\n\
a -- %-20.0f      e -- %-20.0f\n\
b -- %-20.0f      f -- %-20.0f\n\
c -- %-20.0f      g -- %-20.0f\n\
d -- %-20.0f      h -- %-20.0f\n\n",
a,e,b,f,c,g,d,h);

p = t*pm1 + pm2; /* update convergents */
q = t*qm1 + qm2;
pm2 = pm1;
qm2 = qm1;
pm1 = p;
qm1 = q;

```

```

        (void) printf("\
New approximation: %20.16e\n\
True result:      %20.16e\n\n",
        p/q,vz);
    }
}
}
}

```

```

double rz(u,v) /* Written to evaluate correctly with */
double u,v;    /* POSINF as one or both arguments */
{
return((a + b/v + c/u + d/(u*v))/(e + f/v + g/u + h/(u*v)));
}

```

/* Code for inputs */

```

double x()
{
return(2);
}

```

```

double rx()
{
double sqrt();

return(1.0 + sqrt(2.0));
}

```

```

double y()
{
return(2);
}

```

```

double ry()
{
double sqrt();

return(1.0 + sqrt(2.0));
}

```

I.2 Generalized Continued Fractions

```

#include <stdio.h>

unsigned oldstatus,newstatus,fpstatus_();

double a,b,c,d,e,f,g,h;
double p,q,pm1,qm1,pm2,qm2;
double min,max,deltax,deltay;

double POSINF,value[3];

main()
{
void ingestx(),ingesty(),outputz(),getlims();
double vx,vy,vz,rx(),ry(),rz();

newstatus = 0; /* for clearing program status flags */

vx = 0.0; /* for finding limit values */
POSIINF = 1/vx;
value[0] = -1.0;
value[1] = 1.0;
value[2] = POSINF;

(void) fprintf(stderr,"initialize coefficient cube:\n");
(void) fprintf(stderr,"a: ");

```



```

(void) scanf("%lf",&a);
(void) fprintf(stderr,"b: ");
(void) scanf("%lf",&b);
(void) fprintf(stderr,"c: ");
(void) scanf("%lf",&c);
(void) fprintf(stderr,"d: ");
(void) scanf("%lf",&d);
(void) fprintf(stderr,"e: ");
(void) scanf("%lf",&e);
(void) fprintf(stderr,"f: ");
(void) scanf("%lf",&f);
(void) fprintf(stderr,"g: ");
(void) scanf("%lf",&g);
(void) fprintf(stderr,"h: ");
(void) scanf("%lf",&h);

vx = rx();      /* prepare values for descriptive output */
vy = ry();
vz = rz(vx,vy);

pm2 = 0;        /* initialize convergents */
qm2 = 1;
pm1 = 1;
qm1 = 0;

ingestx();
ingesty();

while(1) {
    getlims();
    if(max - min < 0.5) {
        outputz();

        (void) printf("\
Coefficient cube after output:\n\n\
a -- %-20.0f  e -- %-20.0f\n\
b -- %-20.0f  f -- %-20.0f\n\

```

```

c -- %-20.0f g -- %-20.0f\n\
d -- %-20.0f h -- %-20.0f\n\n",
    a,e,b,f,c,g,d,h);
    (void) printf("\
New approximation: %20.16e\n\
True result:      %20.16e\n\n",
    p/q,vz);
    }
else {
    if(deltax >= deltay)
        ingestx();
    else
        ingesty();
    }
}
}

```

```

void ingestx()
{
double t,ta,tb,tc,td,te,tf,tg,th;
double x();

t = x();

oldstatus = fpstatus_(&newstatus); /* clear flags */

ta = a;      /* update coefficient values */
tb = b;
tc = c;
td = d;
te = e;
tf = f;
tg = g;
th = h;

a = t*ta + tc;

```

```

b = t*tb + td;
c = ta;
d = tb;
e = t*te + tg;
f = t*tf + th;
g = te;
h = tf;

oldstatus = fpstatus_(&newstatus); /* test flags */
if(oldstatus & 512) {
    (void) printf("\n\
    Inexact update ingesting x partial quotient %1.0f\n",
    t);
    exit(1);
}
(void) printf("\
    Ingested the x partial quotient %1.0f\n",t);
}

void ingesty()
{
double t,ta,tb,tc,td,te,tf,tg,th;
double y();

t = y();

oldstatus = fpstatus_(&newstatus);

ta = a;      /* update coefficient values */
tb = b;
tc = c;
td = d;
te = e;
tf = f;
tg = g;
th = h;

```

```

a = t*ta + tb;
b = ta;
c = t*tc + td;
d = tc;
e = t*te + tf;
f = te;
g = t*tg + th;
h = tg;

oldstatus = fpstatus_(&newstatus);
if(oldstatus & 512) {
    (void) printf("\n\
    Inexact update ingesting y partial quotient %1.0f\n",
        t);
    exit(1);
}
(void) printf("\
    Ingested the y partial quotient %1.0f\n",t);
}

```

```

void outputz()
{
    double t,ta,tb,tc,td,te,tf,tg,th;
    double ceil(),floor();

    t = (min+max)/2.0;
    ta = floor(t);
    tb = ceil(t);

    if(t - ta < tb - t)
        t = ta;
    else
        t = tb;

    oldstatus = fpstatus_(&newstatus);
}

```

```

ta = a;      /* update coefficient values */
tb = b;
tc = c;
td = d;
te = e;
tf = f;
tg = g;
th = h;

a = te;
b = tf;
c = tg;
d = th;
e = ta - t*te;
f = tb - t*tf;
g = tc - t*tg;
h = td - t*th;

oldstatus = fpstatus_(&newstatus);
if(oldstatus & 512) {
    (void) printf("\n\
    Inexact update outputting partial quotient %1.0f\n",
    t);
    exit(1);
}
(void) printf("\n\
    Output the partial quotient %1.0f\n\n",
    t);

p = t*pm1 + pm2; /* update convergents */
q = t*qm1 + qm2;
pm2 = pm1;
qm2 = qm1;
pm1 = p;
qm1 = q;
}

```

```

void getlims()
{
int i,j;
double temp,limits[3][3],minx,maxx,miny,maxy,rz();

min = POSINF;
max = -POSINF;
for(i=0; i<3; ++i) {
    for(j=0; j<3; ++j) {
        temp = rz(value[i],value[j]);
        limits[i][j] = temp;
        if(temp < min)
            min = temp;
        if(temp > max)
            max = temp;
    }
}

deltax = 0.0;
for(j=0; j<3; ++j) {
    minx = POSINF;
    maxx = -POSINF;
    for(i=0; i<3; ++i) {
        temp = limits[i][j];
        if(temp < minx)
            minx = temp;
        if(temp > maxx)
            maxx = temp;
    }
    temp = maxx - minx;
    if(temp > deltax)
        deltax = temp;
}

deltay = 0.0;

```

```

for(i=0; i<3; ++i) {
    miny = POSINF;
    maxy = -POSINF;
    for(j=0; j<3; ++j) {
        temp = limits[i][j];
        if(temp < miny)
            miny = temp;
        if(temp > maxy)
            maxy = temp;
    }
    temp = maxy - miny;
    if(temp > deltay)
        deltay = temp;
}
}

```

```

double rz(u,v)    /* Written to evaluate correctly with */
double u,v;      /* +- POSINF as one or both arguments */
{
    return((a + b/v + c/u + d/(u*v))/(e + f/v + g/u + h/(u*v)));
}

```

/* Code for inputs */

```

double x()
{
    return(2);
}

```

```

double rx()
{
    double sqrt();

    return(1.0 + sqrt(2.0));
}

```

```

double y()
{
return(2);
}

double ry()
{
double sqrt();

return(1.0 + sqrt(2.0));
}

```

I.3 Sample Modifications

```

double x()
{
static int index = 0;
double q;

if(index == 0)
    q = 2.0;
else if(index == 1)
    q = 1.0;
else {
    if((index-2)%3 == 0)
        q = 2.0*(1.0 + (((double) index) - 2.0)/3.0);
    else
        q = 1.0;
}
++index;
return(q);
}

```



```
double rx()  
{  
double exp();  
  
return(exp(1.0));  
}
```

RADC/COTC 10
ATTN: Jon B. Valente
Griffiss AFB NY 13441-5700

Odyssey Research Assoc., Inc. 5
301A Harris B. Dates Drive
Ithaca NY 14850-1313

RADC/DOVL 1
Technical Library
Griffiss AFB NY 13441-5700

Administrator 2
Defense Technical Info Center
DTIC-FDA
Cameron Station Building 5
Alexandria VA 22304-6145

Strategic Defense Initiative Office 2
Office of the Secretary of Defense
Wash DC 20301-7100

AFCSA/SAMI 1
ATTN: Miss Griffin
10363 Pentagon
Washington DC 20330-5425

HQ USAF/SCTT 1
Pentagon
Washington DC 20330-5190

SAF/AQSC 1
Pentagon 4D-267
Washington DC 20330-1000

HQ AFSC/XTKT 1
Andrews AFB DC 20334-5000

HQ AFSC/XTS Andrews AFB MD 20334-5000	1
HQ AFSC/XRK Andrews AFB MD 20334-5000	1
HQ SAC/SCPT OFFUTT AFB NE 68113-5001	1
DTESA/RQE ATTN: Mr. Larry G. McManus Kirtland AFB NM 87117-5000	1
HQ TAC/DRIY ATTN: Mr. Westerman Langley AFB VA 23665-5001	1
HQ TAC/DOA Langley AFB VA 23665-5001	1
HQ TAC/DRCA Langley AFB VA 23665-5001	1
ASD/AFALC/AXAE ATTN: W. H. Dungey Wright-Patterson AFB OH 45433-6533	1
WRDC/AAAI Wright-Patterson AFB OH 45433-6533	1
AFIT/LDEE Building 640, Area B Wright-Patterson AFB OH 45433-6583	1

WRDC/MLTE Wright-Patterson AFB OH 45433	1
WRDC/FIES/SURVIAC Wright-Patterson AFB OH 45433	1
AAMRL/HE Wright-Patterson AFB OH 45433-6573	1
AFHRL/OTS Williams AFB AZ 85240-6457	1
AUL/LSE Maxwell AFB AL 36112-5564	1
HQ Air Force SPACECOM/XPYS ATTN: Dr. William R. Matoush Peterson AFB CO 80914-5001	1
Defense Communications Engr Center Technical Library 1860 Wiehle Avenue Reston VA 22090-5500	1
C3 Division Development Center Marine Corps Development & Education Command Code DIOA Quantico VA 22134-5080	2
US Army Strategic Defense Command DASD-H-MPL PO Box 1500 Huntsville AL 35807-3801	1

Commanding Officer Naval Avionics Center Library D/765 Indianapolis IN 46219-2189	1
Commanding Officer Naval Ocean Systems Center Technical Library Code 96423 San Diego CA 92152-5000	1
Commanding Officer Naval Weapons Center Technical Library Code 3433 China Lake CA 93555-6001	1
Superintendent Naval Post Graduate School Code 1424 Monterey CA 93943-5000	1
Commanding Officer Naval Research Laboratory Code 2627 Washington DC 20375-5000	2
Space & Naval Warfare Systems COMM PMW 153-3DP ATTN: R. Savarese Washington DC 20363-5100	1
Commanding Officer US Army Missile Command Redstone Scientific Info Center AMSMI-RD-CS-R (Documents) Redstone Arsenal AL 35898-5241	2
Advisory Group on Electron Devices Technical Info Coordinator ATTN: Mr. John Hammond 201 Varick Street - Suite 1140 New York NY 10014	2
Los Alamos Scientific Laboratory Report Librarian ATTN: Mr. Dan Baca PO Box 1663, MS-P364 Los Alamos NM 87545	1
Rand Corporation Technical Library ATTN: Ms. Doris Helfer PO Box 2133 Santa Monica CA 90406-2139	1

USAG
ASH-PCA-CRT
Ft. Huachuca AZ 85613-6000

1

1339 EIG/EIET
ATTN: Mr. Kenneth W. Irby
Keesler AFB MS 39534-6343

1

JTFPO-TD
Director of Advanced Technology
ATTN: Dr. Raymond F. Freeman
1500 Planning Research Drive
McLean VA 22102

1

HQ ESC/CWPP
San Antonio TX 78243-5000

1

AFEWC/ESRI
San Antonio TX 78243-5000

3

485 EIG/EIR
ATTN: M Craft
Griffiss AFB NY 13441-6348

1

ESD/XTP
Hanscom AFB MA 01731-5000

1

ESD/AVSE
Building 1704
Hanscom AFB MA 01731-5000

2

HQ ESD SYS-2
Hanscom AFB MA 01731-5000

1

Director
NSA/CSS
T513/TDL
ATTN: Mr. David Marjarum
Fort George G. Meade MD 20755-6000

1

Director
NSA/CSS
W156
Fort George G. Meade MD 20755-6000

1

Director
NSA/CSS
DEFSMAC
ATTN: Mr. James E. Hillman
Fort George G. Meade MD 20755-6000

1

Director
NSA/CSS
R5
Fort George G. Meade MD 20755-6000

1

Director
NSA/CSS
R8
Fort George G. Meade MD 20755-6000

1

Director
NSA/CSS
S21
Fort George G. Meade MD 20755-6000

1

Director
NSA/CSS
R523
Fort George G. Meade MD 20755-6000

2

SDI/S-P1-BM
ATTN: Cmdr Korajo
The Pentagon
Wash DC 20301-7100

1

SDIO/S-PL-3M
ATTN: Capt Johnson
The Pentagon
Wash DC 20301-7000

1

SDIO/S-PL-3M
ATTN: Lt Col Rindt
The Pentagon
Wash DC 20301-7100

1

IDA (SDIO Library)
ATTN: Mr. Albert Perrella
1801 N. Beauregard Street
Alexandria VA 22311

1

SAF/AQSD
ATTN: Maj M. K. Jones
The Pentagon
Wash DC 20330

1

AFSC/CV-D
ATTN: Lt Col Flynn
Andrews AFB MD 20334-5000

1

HQ SD/XR
ATTN: Col Heimach
PO Box 92960
Worldway Postal Center
Los Angeles CA 90009-2960

1

HQ SSD/CNC
ATTN: Col O'Brien
PO Box 92960
Worldway Postal Center
Los Angeles CA 90009-2960

1

HQ SD/CNCI
ATTN: Col Collins
PO Box 92960
Worldway Postal Center
Los Angeles CA 90009-2960

1

HQ SD/CNCIS 1
ATTN: Lt Col Pennell
PO Box 92960
Worldway Postal Center
Los Angeles CA 90009-2960

ESD/AT 1
ATTN: Col Ryan
Hanscom AFB MA 01731-5000

ESD/ATS 1
ATTN: Lt Col Oldenberg
Hanscom AFB MA 01731-5000

ESD/ATN 1
ATTN: Col Leib
Hanscom AFB MA 01731-5000

AFSTC/XPX (Lt Col Detucci) 1
Kirtland AFB NM 87117

USA SDC/DASD-H-SB (Larry Tubbs) 1
P. O. Box 1500
Huntsville AL 35807

AFSPACECOM/XPB 1
ATTN: Maj Roger Hunter
Peterson AFB CO 80914

GE SDI-SEI 1
ATTN: Mr. Ron Marking
1787 Century Park West
Bluebell PA 19422

MITRE Corp 1
ATTN: Dr. Donna Cuomo
Bedford MAS 01730

SSD/CNI 1
ATTN: Lt Col Joe Rouge
P. O. Box 92960
Los Angeles AFB CA 90009-2960

NTB JPO ATTN: Maj Don Ravenscroft Falcon AFB CO 80912	1
Ford Aerospace Corp c/o Rockwell International ATTN: Dr. Joan Schulz 1250 Academy Park Loop Colorado Springs CO 80910	1
Essex Corp ATTN: Dr. Bob Mackie Human Factors Research Div 5775 Dawson Ave Goleta CA 93117	1
Naval Air Development Ctr ATTN: Dr. Mort Metersky Code 30D Warminster PA 189974	1
RJO Enterprises ATTN: Mr. Dave Israel 1225 Jefferson Davis HWY Suite 300 Arlington VA 22202	1
GE SDI SEI ATTN: Mr. Bill Bensch 1707 Century Park West Bluebell PA 19422	1
HQ AFOTEC/OAHS ATTN: Dr. Samuel Charlton Kirtland AFB NM 87117	1
ESD/XTS ATTN: Lt Col Joseph Toole Hanscom AFB MA 01731	1
SDIO/ENA ATTN: Col R. Worrell Pentagon Wash DC 20301	1
USA-SDC CSSD-H-S&E ATTN: Mr. Doyle Thomas Muntsville AL 35807	1

HQ AFSPACECOM/DOXP
ATTN: Capt Mark Terrace
Stop 7
Peterson AFB CO 80914

1

BBN Systems & Technology
ATTN: Dr. Dick Pew
70 Fawcett St
Cambridge MA 02138

1

ESD/XTI
ATTN: Lt Col Paul Monico
Hanscom AFB MA 01730

1

CSSD-H-SB
ATTN: Mr. Larry Tubbs
Commander USA SDC
PO Box 1500
Huntsville AL 35807

1

USSPACECOM/J5B
ATTN: Lt Col Harold Stanley
Peterson AFB CO 80914

1

NTS JPO
ATTN: Mr. Nat Sojourner
Falcon AFB CO 80912

1

RADC/COT
ATTN: Mr. Ronald S. Raposo
Griffiss AFB NY 13441

1

Bonnie McDaniel, MDE
313 Franklin St
Huntsville AL 35801

1

The Aerospace Corporation
ATTN: Mr. George Gilley
ML-046
PO Box 92957
Los Angeles CA 90530

5

AF Space Command/XPXIS
Peterson AFB CO 80914-5001

1

AFJTEC/XPP
ATTN: Capt Wrobel
Kirtland AFB NM 87117

1

Director NSA (V43)
ATTN: George Hoover
9800 Savage Road
Ft George G. Meade MD 20755-6000

1

SSD/CNIR
ATTN: Capt Brandenburg
PO BOX 92960-2960
LOS ANGELES CA 90009-2960

1

National Computer Security Center
ATTN: C4/TIC
9800 Savage Road
Fort George G Meade MD 20755-6000

1

Unisys Corp
ATTN: Lorraine D. Martin
5151 Camino Ruiz
Camarillo CA 93011-6004

1

Harris Corp
Government Info Sys Division
ATTN: Ronda Henning
PO Box 98000
Melbourne FL 32902

1

Ford Aerospace & Comm Corp
ATTN: Peter Bake (Mail Stop 29A)
10440 State Highway 83
Colorado Springs CO 80908

1

Mitre Corp
ATTN: Dale M. Johnson (MS 8325)
Burlington Rd
Bedford MA 01730

1

Secure Computing Technology Corp
ATTN: J. Thomas Haigh
2355 Anthony Lane South (Suite 130)

1

St Anthony MN 55418

SRI International
Computer Science Lab
ATTN: Terea Lunt
333 Ravenswood Ave
Menlo Park CA 94025

1

Mitre Corp ATTN: Joshua Guttman (MS 040) Burlington Rd Bedford MA 01730	1
Mitre Corp ATTN: Javier Thayer (MS A040) Burlington Rd Bedford MA 01730	1
Computational Logic, Inc. ATTN: Dr. Donald I. Good 1717 W. 6th St (Suite 290) Auston TX 78703	1
Odyssey Research Assoc. ATTN: Dr. Richard Platek 301A Harris B. Dates Dr. Ithaca NY 14850-1313	1
National Security Agency ATTN: Larry Patch/R5 9800 Savage Rd Ft Meade MD 20755	1
National Computer Security Center ATTN: Mark Woodcock/C33 9800 Savage Rd Ft Meade MD 20755	1
Naval Research Laboratory ATTN: Carl E. Lardwehr (Code 7593) Wash DC 20375	1
US Army CECOM/CENTACS AMSEL-RD-COM-TC-2 ATTN: John W. Preusse Ft Monmouth NJ 07703	1
Gemini Computers Inc. ATTN: Roger Schell 60 Garden Court (Suite 110) Monterey CA 93940	1
Boeing Aerospace Co ATTN: Daniel Schnackenberg (RH-35) P.O. Box 3999 Seattle WA 98124	1

BRN Laboratories, Inc. ATTN: Steve Vinter 10 Moulton Street Cambridge MA 02238	1
University of California Computer Science Dept ATTN: Prof Richard A. Kemmerer Santa Barbara CA 93106	1
Institute for Defense Analyses Computer & Software Engineering Div ATTN: William Mayfield 1801 N. Beauregard St Alexandria VA 22311	1
Unisys Corp ATTN: Deborah Cooper (MS 91-11) 2525 Colorado Ave Santa Monica CA 90406-9988	1
Trusted Informations Systems ATTN: Stephen T. Walker 3060 Washington Rd Glenwood MD 21738	1
SRI International Computer Science Lab ATTN: John Rushby 333 Ravenswood Ave Menlo Park CA 94025	1
NASA Langley Research Center ATTN: Ricky Butler (MS 130) Hampton VA 23665-5225	1
National Computer Security Center ATTN: Rob Johnson/C33 9800 Savage Rd Ft Meade MD 20755-6000	1
National Computer Security Center ATTN: Howard Stainer/C3 9800 Savage Rd FT Meade MD 20755-6000	1
SPAWAR/Code 3242 ATTN: Bob Kolacki Wash DC 20363-5100	1

Strategic Defense Initiative Office
Office of the Secretary of Defense
Wash DC 20301-7100

1

National Security Agency
ATTN: George Hoover/V45
9800 Savage Rd
Ft Meade MD 20755-6000

1

Naval Research Laboratory
ATTN: John McLean (Code 5540)
Wash DC 20375

1

DARPA/ISTO
ATTN: (Dr. William Scherlis)/DL-10
1400 Wilson Blvd
Arlington VA 22209-2308

1



MISSION of Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C³I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C³I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.